An Empirical study of the JaMoPP project using code coverage based system tests.

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

Colin Bell B.Sc.

January 2010

Supervisor: James Power

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Software Engineering, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____ Date: _____

Abstract

Model Driven Software Development is an attempt to build software systems by combining the two disciplines of software programming and modelling. But due to the gap in technology, there are a number of problems in using these models to generate the final source code implementation. This has meant that Model Driven Software Development has become a future goal rather than a current reality. The Java Model Parser and Printer project tries to bridge this gap by turning Java into a model. This means that Java can now be used and implemented like any of the other models within the Model Driven Software Development process.

This paper will take a look at the effectiveness of the Java Model Parser and Printer project by performing a number of System tests on it. This will help indicate now well the project can transform Java input files to java models. Coverage readings will be taken on these tests to look at how much of the project's grammar and abstract syntax tree are being used during the process of transforming Java source code into Java models. A look will then be taken at the coverage results to see if a correlation exists between the grammar and the abstract syntax tree results. Finally conclusions and future work will be discussed to close the paper.

Contents

Section 1: Introduction 6

Section 2: Background..... 8

2.1 Eclipse Development environment and Framework..... 8

2.2 The EMF Framework and Ecore Modelling..... 9

2.3 Java Model Parser and Printer..... 11

2.4 Java Meta-Model..... 13

2.5 Text Syntax for Java..... 14

2.6 The Complete System View and Modelling process.... 17

2.7 Software Testing and Code Coverage 18

2.8 Goals of the paper 19

Section 3: Testing Approach 20

3.1 JaMoPP setup and package descriptions 21

3.2 Testing process overview 23

Section 4: Testing and Coverage Results..... 26

Section 5: Conclusion..... 35

5.1 Future Work..... 37

References 38

Table of Figures

Figure 1: EMF Model as the common high level representation within the EMF framework [3] 9
Figure 2: A subset of the Ecore meta-model [3] 10
Figure 3: A subset of the Java meta-model defined in the JaMoPP project [1]. 14
Figure 4: The process of creating an ANTLR Parser from a concrete syntax after a meta-model has been defined within the EMF framework [5] 15
Figure 5: The full process that was used in generating the Java specific parser for the JaMoPP project[1] 16
Figure 6: The complete JaMoPP project and modelling process [1] 17
Figure 7: A Java class beside its Java meta-model instance which has the structure of an Ecore model [4] 25
Figure 8: Partial view of the subfolders of the generated folder found in the org.emftext.language.java package. 27
Figure 9: Break down of the Coverage results for the Parser methods that map to the corresponding impl folders of the org.emftext.language.java package. The names of the folders correspond to elements of the Java language. 28
Figure 10: Graph of the coverage results for the parser and AST for the modifier component of the Java language. 29
Figure 11: Coverage results for the All Test tests for the parser and the AST. 31
Figure 12: The breakdown of the AST results for a blank input file. The total AST result was 24.6%. 32
Figure 13: The breakdown of the parser and AST coverage results for the Netbeans test. 34

Section 1: Introduction

This paper describes a study that uses the Java Model Parser and Printer project. Its place within the world of software design and software design technology will be described. It will be evaluated in a serious of system tests that will indicate its effectiveness in handling the Java language.

Model driven software development (MDSD) methodologies are used in the production of software systems that start out as a process of representing each stage of the system as a model or abstraction. These models initially are more closely related to the particular domain that the final product is been created for, rather than software programming focused models. This allows for a number of advantages which would include maximised compatibility between systems and also allowing the development team and final end users to communicate about the system more clearly. This form of software development is considered to be effective if the models used to represent the system can provide ease of understanding to the high level modellers and the final stakeholders while providing enough detail for the low level coding implementation of the final system.

The MDSD methodology combines the two disciplines of coding and modelling. Practitioners of these two disciplines view their own discipline as being completely different and separate from the other. Due to this view being upheld, it has had a knock on effect in the world of MDSD. The MDSD process is designed to be able to create a full software system. Starting from an initial system model and then generating other models at different levels of abstraction all the way down to the final code implementation. But this final transformation is done in a weak structured manner [1] while the other model transformations are done in a well defined way. Structure is very important within the world of modelling and any loss of structure will have undesired consequences within transformations between the models.

This has been the state of MDSD until recently when a push to unite the two disciplines has started to occur. The Eclipse development environment is a piece of software that allows for software development for the Java language and many other languages. It has become very popular within the software community due to its extendable architecture and open source nature. A modelling plug-in for Eclipse, known as the Eclipse Modelling Framework (EMF), extends Eclipse to handle models such as UML. This framework itself can also be extended to allow it to handle any standardised model. The EMF plug-in can take

a Java source code file and transform it into its corresponding UML representation and also into a XML schema. While the EMF plug-in can transform Java into UML and back again it doesn't treat Java as a modelling language but rather still just as code. This means that Java is not handled like other EMF models and this still leaves the gap between modelling and coding open.

A group of researchers from the Technical University Dresden in Germany identified the gap mentioned above and looked for ways to turn Java into a full modelling language. They used the EMF plug-in as its framework and architecture. They came up with the Java Model Parser and Printer (JaMoPP) project which supplies a parser for turning a Java source code file into a Java EMF model and then a printer which can take a Java EMF model and turn it back into source code.

This paper will look at the JaMoPP project and test its effectiveness. From these tests some coverage readings will be collected. The coverage results will provide information on how much of the grammar has been used and also the percentage of coverage achieved for the corresponding generated abstract syntax tree. Areas where coverage has been fully achieved can be identified and also areas where coverage has not fully been achieved or where it has only been particularly done can also be identified. From the results it can also be seen if there is a correlation between the coverage in the parser and the abstract syntax tree. There have been many studies conducted over the years on the coverage results that a set of input tests achieve for a language's grammar. Many papers look at grammar coverage from a number of possible perspectives. One method is known as rule based coverage which is similar to decision coverage at the code level in a standard software testing context [2]. This paper will look at JaMoPP's grammar using statement coverage which is getting the percentage of the lines of code that are executed during testing. This paper is unique in the fact that it is testing a grammar that transforms Java code to a Java meta-model instance which can then be transformed and handled like any EMF model.

In section 2 the background for the paper will be laid out. This will cover Eclipse, the EMF plug-in, the JaMoPP plug-in, software testing as it relates to this paper and also the goals of this paper. Section 3 will describe the testing process used in this paper. It will identify test suits as well as the packages and classes of the JaMoPP project that will be used for gathering coverage results. Section 4 will discuss the results achieved from the tests and

also provide information on which paths through the grammar and the abstract syntax tree were executed. Section 5 will conclude the paper.

Section 2: Background

This section will describe all the background details necessary to understand the JaMoPP project and its goals. It will describe the Eclipse project and its plug-in capacity including the EMF Framework plug-in with its core language Ecore. It will describe how the JaMoPP developers defined their Java meta-model and the need for a concrete syntax so that the meta-model can be of use in a practical sense. Also this section will describe the process of creating an instance of the Java meta-model and how it slots into the EMF Framework. Finally a brief description of software testing and testing techniques will be given with a focus on their relevance to this paper.

2.1 Eclipse Development environment and Framework

Eclipse is an open source software development project that is made available under the common public license initiative run by IBM. It supports the practice of software development in multiple languages and also can be run on numerous operating systems. It is comprised of an Integrated Development Environment (IDE) with an extendable architecture. It has a small run-time kernel and all other functionality is provided by plug-ins into its architecture. This is in contrast to other development environments which all have hard coded functionality which also makes Eclipse a lightweight program.

Eclipse is divided into three main projects. First is known as the Eclipse project which contains the core components required for development with Eclipse. These components are fixed and are commonly downloaded as the Eclipse Software Development Kit. This project itself divides down into three component projects [3], which are as follows:

1. The Eclipse platform: This is a framework used to build IDE's. It simply defines the basic structure of an IDE and when specific tools are used to extend the framework this will define the particular IDE for the language that is required [3].
2. Java Development Tools (JDT): This is a Java tool set that expands the Eclipse platform above to allow for a development environment to develop Java programs.

3. Plug-in Development Environment (PDE): extends the JDT by providing tools to handle the non Java aspects of plug-in development. One example would be providing registration for plug-in extensions.

The second project is known as The Tools Project. This project defines and coordinates the integration of different frameworks and other tool sets for defining IDE's for other languages. This includes model based frameworks i.e. the EMF Framework and also other IDE's such as the CDT for C/C++ and many other languages. The final project is known as The Technology Project. This project provides the opportunity for researches and academics to get involved in the evolution of the eclipse project [3].

2.2 The EMF Framework and Ecore Modelling

The EMF framework is a modelling framework for eclipse. The EMF project provides a framework and code generation facility that allows for the generation of UML, XML or Java implementations of a system [3]. Regardless of which format the system is defined in an EMF model acts as the common high level representation that holds them all together.

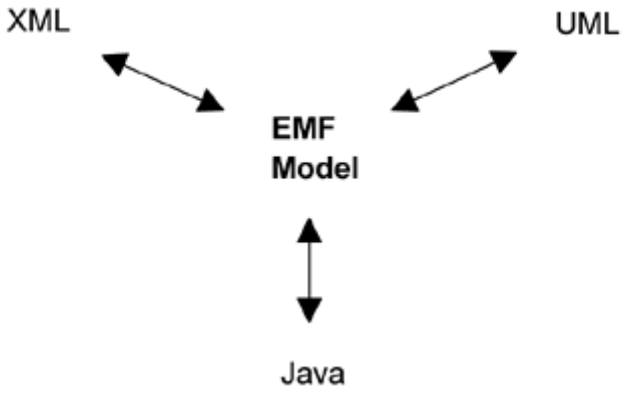

Figure 1: EMF Model as the common high level representation within the EMF framework [3]

For example imagine a system that implements a library book catalogue. The system has been coded in Java and by only clicking a button the corresponding UML diagram can be generated or an XML schema implementation can be generated.

EMF can be envisioned as the start of the practical realisation of the combination of the two disciplines of modelling and coding. An EMF model is at the same level of abstraction as the class diagram subset of UML [3]. This is enough a level of abstraction for a number of benefits to occur for programmers and modellers. Some of these benefits would

be providing understandability for both programmers and modellers, data integration between applications and also allows for the auto-generation of code and models. EMF is integrated with and fine tuned for efficient programming [3]. This brings together high level modelling with low level programming [3] which is one key area that the JaMoPP project wants to build on.

EMF is described in an XMI model specification and provides a tool set known as the MDT (Model Development tools) plug-in for Eclipse to provide support for adapter classes and editors for models within Eclipse and all this was developed using Java. The EMF framework needs a standardised well established meta-model to allow for the creation of models, meta-models and the auto-generation of code. EMF uses Ecore which is a well known meta-language to achieve this. Ecore is called a core language because it itself is also an EMF model which makes Ecore a meta-meta-model.

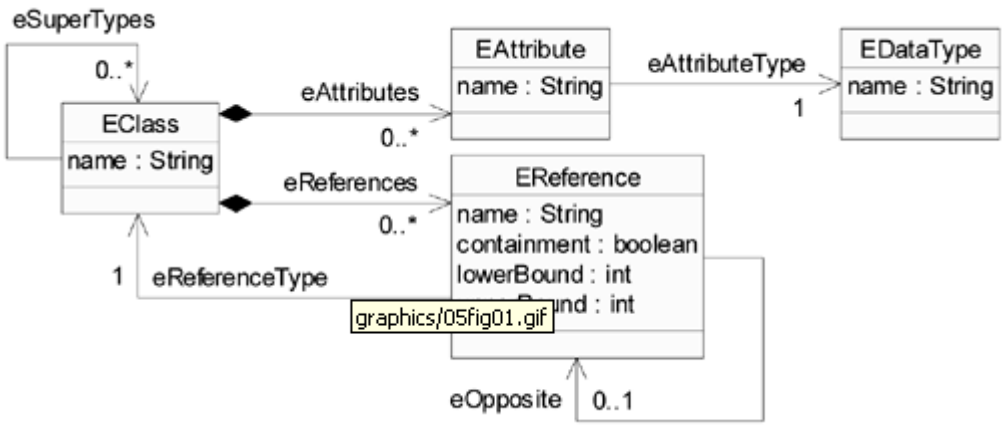

Figure 2: A subset of the Ecore meta-model [3]

Ecore was used by the JaMoPP developers to create their Java meta-model and with this they gained the advantage of been able to use Java like any other EMF model. The above diagram is a simplified subset of the process that occurs when an Ecore model is been created. It shows some of the major classes of the Ecore language used in building a model from Ecore but it should be noted that the there are many more relationships and other classes involved in the Ecore meta-modelling process. A brief description of some of the more major aspects and classes of Ecore will be given.

- The EClass class is used to model classes. These classes are identified by a unique identifier and Ecore can also represent data about a class. These data components of

a class can represented using EAttributes and EReferences. To support inheritance, a class can refer to a number of other classes as its super types [3].

- The EAttribute class is used to model attributes which represent the attributes of the class and each of these have specific EDataTypes which need to be handled by the potential Java meta-model in order to faithfully create a instance of the Java meta-model without losing information.
- The EReference class is used for modelling associations between classes which has to be of type EClass.
- The EDataType class specifies attributes to model primitive types and object data types which are explicit to Java and Java models but would not be already defined within the EMF framework.
- Related EClasses and EDataTypes are grouped into packages called EPackage.
- The EFactory class is used to create instances of the EClasses and values of the EDataTypes that belong to the EPackages.

There are further structural, types, behavioural and classifiers classes involved in building a model from the Ecore meta-modelling language but are too many to mention in order to provide a brief discussion of the Ecore meta-model and how the JaMoPP developers would have used the Ecore language within their work.

When the JaMoPP developers used the Ecore language to define their Java meta-model this is how the Java language would map onto the Ecore classes. EClasses would be used to represent Java interfaces. The EAttributes and ERferences would be identified from the methods that are defined in each of the interfaces. An Ecore class known as EEums would be used to model Java classes. EPackages could also be used to model interfaces but their specific components would come from the files import statements. EDataTypes may be specified explicitly or can be mapped directly to corresponding Java types. Special comments are used by the EMF framework to identify model elements and they can also provide additional information that is not directly expressed in the Java interfaces [17].

2.3 Java Model Parser and Printer

As mentioned in the introduction section, the JaMoPP developers found a gap within the MDSD process between the final stages of moving from the models to the concluding code of the system. The code that is outputted is normally generated as plain text and loses the

structure that was present in the models thanks to the well defined and formal meta-modelling process. Losing structure causes many issues and problems within the modelling world. For one there would be no guarantees regarding syntactic or semantic correctness [4] in the generated model of the code. Also during model generation, if errors were to occur, there would be no way for the software to trace back and find the element of the model that caused the generation of that particular line of code.

The goal of Model-Driven Software Development is the (semi-)automatic generation of software systems from models across multiple stages [1]. Basically any Model that is created to represent the system no matter the level of abstraction should be able to be used to help generate any other model that represents the system at a different level of abstraction all the way down to the Java source code. This is achieved using a standardised model meta-language. A meta-language defines the types and constraints for all the models and the transformations between models within the MDSD process. This allows for all transformations to be checked for correctness. These models which represent the potential software system should be standardised, have the possibility to be redefined and finally transformed into other models.

This is where the JaMoPP project comes in to play. The developers of the JaMoPP project identified the above gap in the MDSD process and argued that it can be fixed. They see that people view the modelling process and the coding process as two completely separate disciplines and that if this view point could be changed the gap could be removed. They argue that modelling tools could be developed that could handle the Java language in the same manner that the current tools handle modelling languages. Then the tools could be used to model Java just like any other modelling language.

The JaMoPP project provides the following three components in order to turn Java into a workable modelling language:

1. JaMoPP defines a complete meta-model for Java that covers the whole of the Java 5 language. The Java meta-model is defined in the commonly used meta-modelling language Ecore. This allows Java to be processed by meta-modelling tools such the EMF framework. [1]
2. JaMoPP defines a text syntax that conforms to the Java language specification. This is used to generate a parser which can be used to create instances of the meta-model

from Java source code and a printer which transforms instances of the meta-model back into Java source code.

3. JaMoPP's Java meta-model was specially designed by the JaMoPP developers to handle Java's referencing and type rules. This is done using generated objects known as Java resolvers. These resolvers guarantee that an instance of the Java meta-model correctly models a Java source file and also Java's static semantics.

The final result of the JaMoPP project is that Ecore based modelling tools can process Java files in the same way that they can process other models.

2.4 Java Meta-Model

The Java Language Specification does not include a complete explicit meta-model, the syntax and semantics of the language are specified either informally or using syntax diagrams [1]. The closest thing to a meta-model that Java has is its build-in reflection methods. These do not capture fine-grained elements like statements and code control flow elements etc [1]. The Javac[18] parser has internal meta-models which are written in Java and help to create the abstract syntax tree which can model all aspects of the Java language but this is not done in a standard meta-modelling language format.

There are a number of standardised meta-models defined for Java but these suffer from incompleteness. The various problems for each of these meta-models range from not providing ways to capture blocks, statements or expressions to being purely tree structured. The tree structured meta-models cannot model static semantics such as identifiers and these are not resolved to their respective elements but only stored as plain strings [1]. This raises problems when trying to uphold consistency between the original model and newly manipulated versions of the model. Some of these meta-models can transform a Java program into another type of model but not vice versa.

The JaMoPP developers could not find a meta-model that both conforms to a well established meta-modelling language, for example Ecore and also fulfils their need for completeness. They decided to examine and compare the existing meta-models, extract commonalities and extend to fully support the Java Language Specification [1].

The meta-model that the JaMoPP project uses to define its Java meta-model is the core language Ecore. Ecore is a core model (meta-meta-model) that acts as its own meta-

model so it can be defined in terms of itself which allows for it to be used as the core meta-model for the EMF framework.

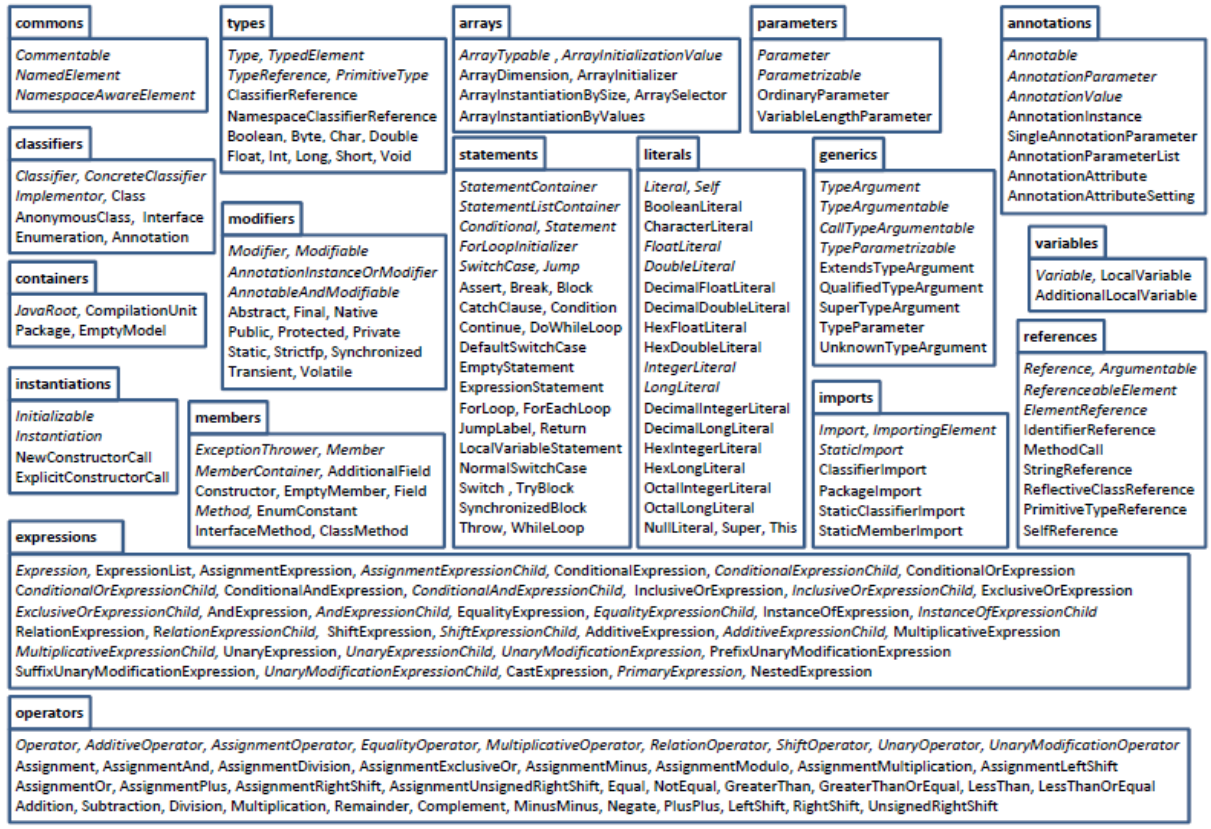

Figure 3: A subset of the Java meta-model defined in the JaMoPP project [1].

The JaMoPP developers ended up creating their meta-model using 80 abstract and 153 concrete classes which are distributed among 18 packages [1]. Their meta-model can model all of Java 5 including its new features, generics and annotations.

2.5 Text Syntax for Java

The current generation of MDSD developers have taken a slight turn away from solely using the conventional method of representing models graphically to now using a usable textual representation for modelling and editing. A number of tools for defining textual syntaxes have a risen over the last few years. Some can produce what is known as Generic syntaxes. These syntaxes are on the same level as meta-modelling languages. This means that a syntax can be derived automatically for concrete modelling languages [4]. Others produce what is known as custom syntaxes that have to be manually specified.

To bridge this gap between these two versions of textual syntax the developers behind the JaMoPP project created EMFText. EMFText allows developers to stepwise refine specifications that are automatically derived from given meta-models. This enables the developer to build custom syntax starting from a generic syntax [4]. EMFText is a plug-in for the Eclipse development environment. This allows for EMFText to work in conjunction with the EMF Framework and also with meta-models defined using EMF.

The EMFText developers used the Extended Backus-Naur Form (EBNF) to form the basis of their syntax specification language ConcreteSyntax (CS). From the CS a Java parser and printer where generated.

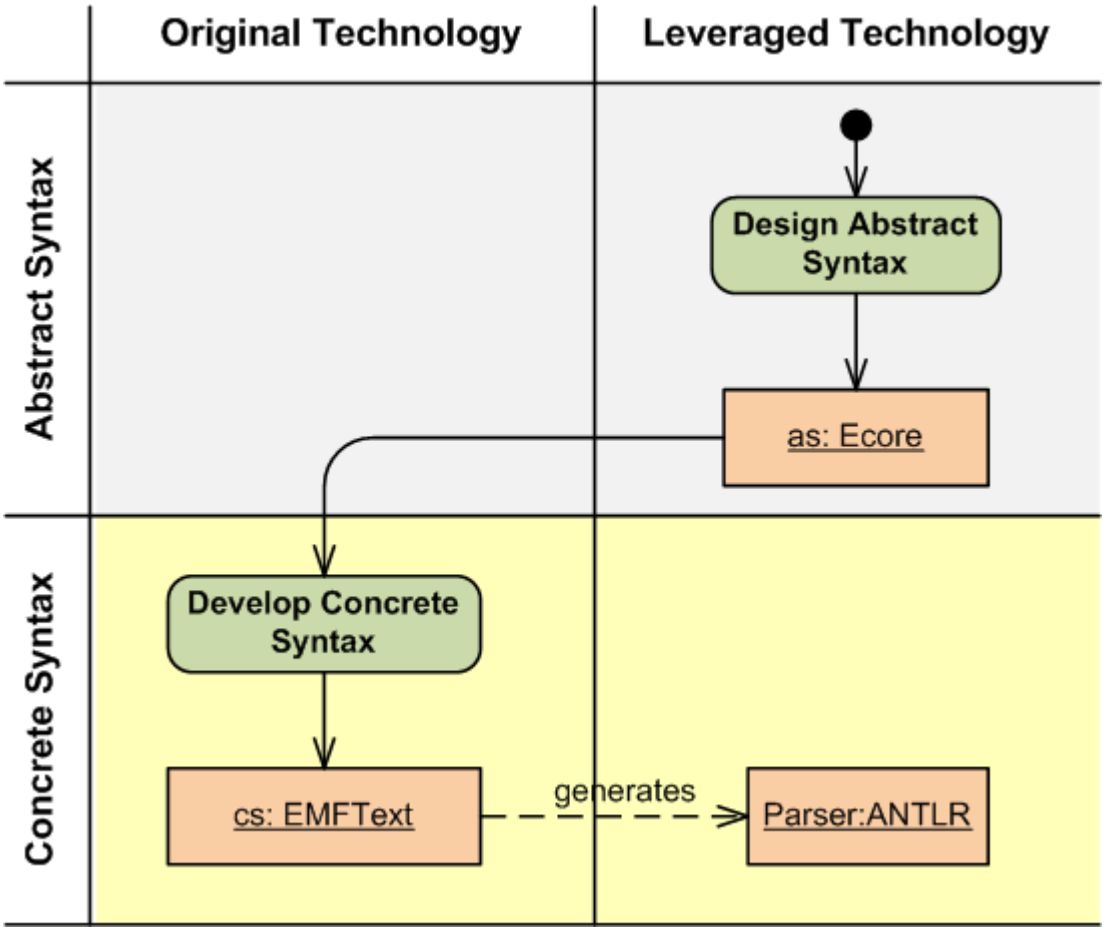

Figure 4: The process of creating an ANTLR Parser from a concrete syntax after a meta-model has been defined within the EMF framework [5]

The diagram above shows the process that the developers of the JaMoPP project used to create the Java meta-model using the EMF framework's core language Ecore and then using that to create the concrete syntax. Using this, as mentioned above EMFText derives a context-free grammar and exports it as an ANTLR [6] parser specification. EMFText then

transparently delegates parser and lexer generation to ANTLR by passing the generated grammar file [7].

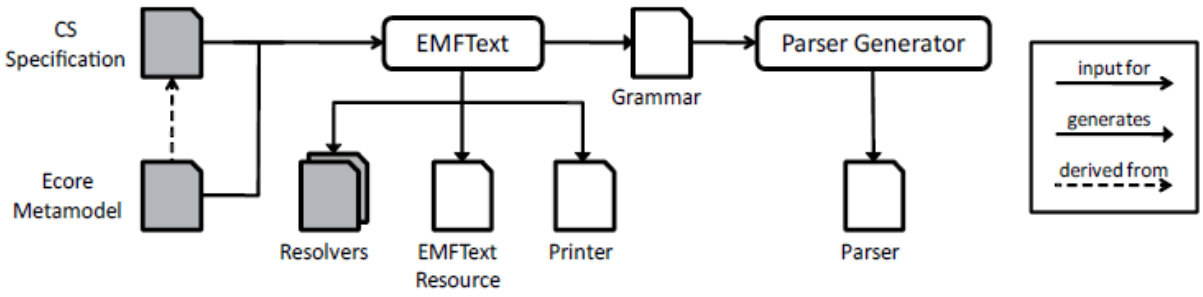

Figure 5: The full process that was used in generating the Java specific parser for the JaMoPP project[1]

Java has scoping and reference rules which need to be modelled correctly by EMFText. EMFText generates objects known as resolvers to take care of these requirements and to convert parsed tokens to an adequate representation of the Java language. The parser is used to create model representations of text based Java programs from the Java meta-model. While the printer performs the opposite operation, it takes an instance of the meta-model and turns it back into a text based representation which is formatted and is in a human readable format. Both instances of the model, whether it's created by the printer or the parser, should be equal and should be able to be transformed between either model without losing or adding in any extra information.

Primarily the JaMoPP developers wanted the project to focus on Java source files but they also included a way to parse from class files. They use the BCEL byte code [8] to create an instance of the Java meta-model but they don't support the use of creating text-based printer models.

2.6 The Complete System View and Modelling process

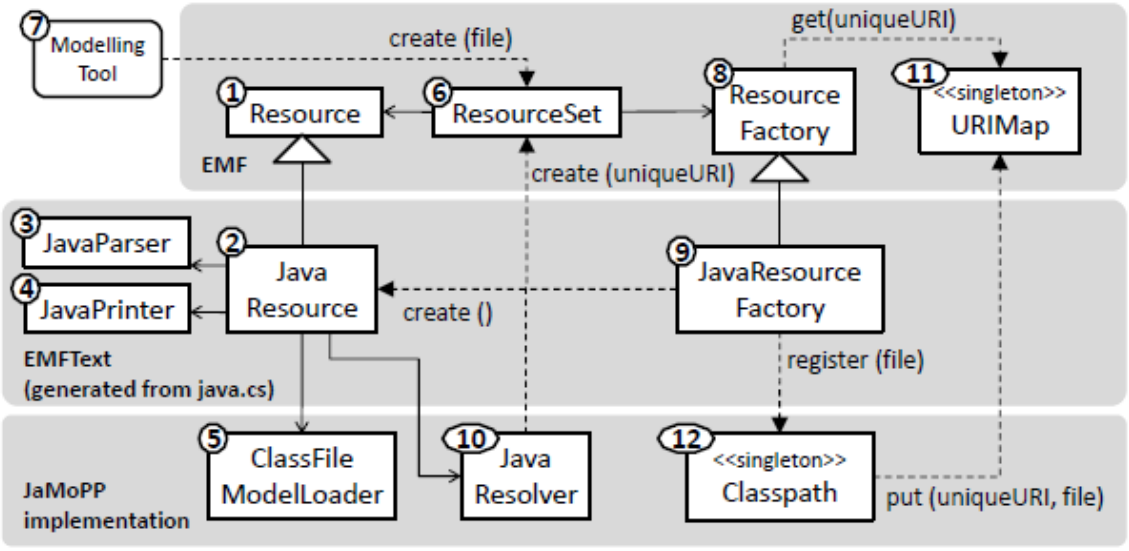

Figure 6: The complete JaMoPP project and modelling process [1]

The above image shows how the JaMoPP project operates when a modelling tool wants to use its Java meta-model. The image also shows the architecture of the project and how it slots into and expands upon the EMF Framework to model Java. The EMF core language, Ecore, brings many benefits to the table by helping EMFText to generate parsers and printers that can fit transparently into the EMF architecture. These generated components can then be used to transform Java into any EMF based model. The Framework can do this regardless of the specific syntax of Java or any other language it is required to model. In what follows, we provide a short description of the events that occur during the process of the EMF Framework transforming a meta-model instance from some model's input with an emphasis on the Java meta-model.

A modelling tool looking to use an EMF meta-model would first connect to the EMF Framework and instantiate a Resource Set [1]. After this the Resource Set acquires a Resource Factory. The file that the modelling tool inputted into the EMF Resource Set is given a unique identifier. This is known as a URI. The URI is usually the location of the file in the operating system. The URI is stored in a global URI map for global resource registry [1]. The URI assists the EMF framework to select the correct factory for the inputted object, in this case the JavaResourceFactory, by looking at the URI's file extension. This process helps to hide the encoding of the resource from the modelling tool that is using the EMF Framework. Next a new specialised resource is generated by EMFText from the Java ConcreteSyntax specification and connects it with a generated parser and a printer [1]. Also

at this point the Java resolver set is generated by EMFText to handle Java specifics such as name/type resolution and reference rules.

Now a new EMF model has been created and this is stored in the system as a resource (Java Resource). This resource handles the parsing, which is known as loading because it loads an instance of the Java meta-model into the EMF Framework, and printing, which is known as saving because it transforms the meta-model back to a textual representation, operations. The final component in the image above is the BCEL class parser. This is implemented in the JaMoPP project and extends the Java parser to handle class files which is outside the scope of this project.

2.7 Software Testing and Code Coverage

Software testing can be described as an empirical investigation into the quality of the software under test. Testing can provide essential information to the developers and the final end user about the effectiveness and operational capabilities of the product. The main purpose of testing is to execute a piece of software with the intent to detect failures in the running of the code. It can be found that not all failures are caused during the process of writing the code but some are caused by non-functional requirements that have not been met. These could be unforeseen security issues and also code performance issues etc. Software testing is a vast area and an essential element in the software development life cycle.

Two of the most important techniques of software testing are known as black box testing and white box testing. Black box testing views a piece of software as a black box and focuses on testing for all the possible input values and the expected output values. It is driven by the specification that was drawn up at the outset of the development life cycle. The major black box testing techniques are equivalence partitioning, boundary value analysis and use case testing. White box testing uses the internal workings of the code and data structures to derive its test cases. The main techniques that are used with this method of testing are branch testing, code coverage, Application Programming Interface (API) testing, path testing and control flow testing.

Code coverage will be used in this paper to evaluate the JaMoPP project. This testing technique is used to measure the degree to which a piece of software has been tested. The area that the coverage criteria covers are as follows:

- Method Coverage- which looks to see if all the methods within the code have been called at least once.
- Statement Coverage- how many lines of code have been executed and how many have not.
- Decision Coverage- Checks to see if each code control structure's path has been tested fully. For example, it checks to see whether a for-loop's true and false paths have both been executed
- Condition Coverage- Checks to see has each Boolean statement been executed for both of its true and false paths.

There are a number of different levels of software testing the one that is of important to this paper is the system test level. System tests are performed on completed and fully integrated software or hardware systems to evaluate how well it lives up to its specification. This phrase of the testing process is normally done through black box testing which has no internal knowledge of the code workings.

2.8 Goals of the paper

The main goal of this paper is to ascertain how well the JaMoPP project can model the Java 5 language. To system test the project to see if it can accept all valid Java 5 programmes that can be thrown at it. To test the project to see can it create a corresponding Java meta-model instance from a valid input file and then retransform the meta-model back to the correct Java source code file. This will prove that it is performing what it is suppose to be able to achieve. The retransformed source file can then be compared to the original input and checked for errors or missing data that was not captured by the JaMoPP project. A code coverage tool will then be used to see how many of the paths in the grammar rules have been covered and also how many of the abstract syntax tree paths have been created during the system tests. These measurements will be then used to see if a correlation exists between the grammar and the abstract syntax tree coverage results. The next two paragraphs will explain a little about grammars and abstract syntax trees to help provide an understanding of their use in programming languages and relevance to the Java meta-model.

Java is a formal language that has a syntax and a grammar. The grammar defines the rules which can be used to form valid strings, which are known as tokens, within the Java language. These tokens are formed from the language alphabet in accordance to the language

syntax. A Java ANTLR lexer, which as mentioned above was created by the CS specification, is used to scan the input code and split the code up into tokens. White space is ignored. These tokens are then fed into the parser which is a context free grammar. The parser will determine if each of these tokens are valid and also that each token appears in the correct order. Formally, a grammar is a four-tuple (*N*, *T*, *S*, *P*) where *N* and *T* are disjoint sets of symbols knows as non-terminals and terminals respectively, *S* is a distinguished element of *N* known as the start symbol, and *P* is a relation between elements of *N* and the union and concatenation of symbols from (*N* U *T*), known as the production rules [2]. Java also has a symbol to represent the empty string. The production rules that are produced describe the valid sentences for the Java language. These sentences are the inputted programs that conform to the grammar of the language [2].

An Abstract syntax tree is the representation of the abstract syntactical structure of the parsed Java input source file. The tree is the result of the path taken through the context free grammar in the parsing stage. The generated abstract syntax tree only shows variables, operations and statements encountered during the parsing of the input file. These are represented as nodes on the tree. The tree does not show all the details of the file and will leave out end of line characters and brackets, which are implied by the visual structure of the tree. Within a standard Java compiler the tree is used for semantic analysis of the code and then for generation of the Java byte code representation of the input file. In the case of the JaMoPP project the abstract syntax tree is a generated model of the code with unresolved cross references [1].

Section 3: Testing Approach

This chapter will briefly describe the process of installing the JaMoPP project within the Eclipse environment. It will describe the packages that are required for the project to correctly work with an aim to add understanding about the system. While describing the packages it will provide an opportunity to identify the classes and sub packages that will be used to gather code coverage measurements. It will identify how the goals of this paper will be achieved. Finally the testing process will be described in detail including the kinds of test that will be run and also ideas on appropriate sources for the tests.

3.1 JaMoPP setup and package descriptions

The JaMoPP project comes as a number of plug-ins for the Eclipse environment. From the JaMoPP webpage [9] there are two ways to install the project into Eclipse. The first way is using the Eclipse download manager to install what is known as the stable JaMoPP project. This is the version that the developers provide people to use their project for model development and practical applications of Java source code to EMF model transformations or from EMF model to Java source code transformations. This version of the software cannot be tested as it does not include their testing base package and other features that are required. All documentation for the project focuses on this version of their software including the project setup demonstration on the website [10]. The other way to install this project is using a subversion tool to access the public repository through Eclipse. The link for the repository is also on the webpage. This version includes the latest builds that the developers are working on and also includes the methodology that they used to test their project on a number of big open sourced projects. This repository does not include the essential SDK of the project nor the base testing package. For this paper, this is the version of the JaMoPP project that will be used to perform the empirical study and test how well the JaMoPP project can model Java.

As mentioned above the JaMoPP project is not well documented and this caused many problems in the correct installation of the project. The closest installation information source is out of date and is only intended for the stable version. The public repository does not hold the complete project or the base testing package. Only by directly contacting the developers where the many issues listed above resolved. This next section of the paper will provide a very brief description of the installation process to add understanding of the JaMoPP project and to also provide information on the packages involved, some of which will be used as the basis for the testing metrics for this paper. The developers of the JaMoPP project were very helpful in finding all the packages and providing the information that is required to get the project fully compiling.

The first set of packages needed for the JaMoPP project, form the core elements of the software and are known as the EMFText SDK. This is the runtime environment plug-ins and ANTLR base code that are used to extend the EMF framework to handle Java the same as any other EMF model. These packages are checked out from a repository into the Eclipse environment and then exported as a “deployable plug-in and fragment project” from Eclipse

into the Eclipse plug-in folder. After this step, the main plug-ins for the project can be checked out of the public repository mentioned earlier. These are the packages that will form the Java specific parser, resolver, and printer components which are generated from the ConcreteSyntax and also specify the Java abstract syntax tree and Java meta-model. There are a total of 5 of these packages.

The org.emftext.commons.antlr3_1_1 package provides the ANTLR runtime mechanisms and source code required by EMFText to extend the ANTLR classes to define a Java specific parser. This is achieved by the ConcreteSyntax which derives a context free grammar and exports it as an ANTLR parser specification to allow the JaMoPP project the power to transform Java source code to instances of the Java meta-model. The ANTLR package comes with four sub-packages only one of which is used to provide components for the JaMoPP project. This sub-package, as mentioned, is extended to provide the base for the parser, printer, lexer, token streams/tokenizer, file readers/ file steams and classes that work on the current token's state. None of the ANTLR package classes will be used as a metric during the testing phase because these are just extended by the org.emftext.langauge.java.resource.java classes.

The org.emftext.language.primitive_types package provides the Java specific primitive types for the Java meta-model. After the package has been checked out of the repository, its .genmodel file has to be run to generate the Java specific EDataTypes for the Ecore meta-model. None of these classes will be used as a metric for our tests as the grammar that will be generated later by the ConcreteSyntax will include the relevant information from these classes.

The org.emftext.langugae.java.resource is a hand written package created by the JaMoPP developers to extend their generated Java parser to handle Java class files and Java byte code. This package will not used as a testing metric.

The org.emftext.langauge.java.resource.java package is the package where the generated parser and printer will be generated to. The sub-packages will also contain the Java resolvers that need to be created to correctly handle Java specific rules. As mentioned above generated sub-packages will hold all the necessary Java specific parser, lexer, printer etc. Code coverage readings will be taken on the Java parser class to see how many of the paths through the Java grammar have been taken as a result of the parsing of the input files. Within the parser class are methods which transform Java statements, generics, primitive types/types

and classifiers etc to the corresponding Java meta-model/Ecore representation. These methods will be recorded for their code coverage and then contrasted against corresponding sub-packages of the Java Abstract syntax tree found in the org.emftext.language.java package.

The org.emftext.language.java package is where the ConcreteSyntax specification is used to generate the Java specific components for the package mentioned in the paragraph above. When the package is checked out of the public repository, its .genmodel file needs to be run first. After this file is run the Java Abstract syntax tree for the Java meta-model will now be generated within newly created sub-packages. As mentioned in the paragraph above these classes will be measured for code coverage and then compared to the corresponding methods in the Java parser class. Next the java.cs file within the package needs to be run. This will finally generated the much talked about grammar and now the entire JaMoPP project has been installed and should now compile fully.

The JaMoPP developers tested their project against big open source software such as Eclipse, JBoss and Netbeans. They have supplied a testing harness and packages in which their tests can be recreated. This is done using Eclipse's built in testing plug-in, Junit [11]. As part of the testing strategy of this paper, these tests will be rerun in chapter 4. They were selected by the developers as a good way to test the majority of the Java meta-model and they will help to gain great code coverage for the majority of both the grammar and the abstract syntax tree which can then be compared against each other.

3.2 Testing process overview

This section will explain the testing process that will be used for testing the JaMoPP system within this paper. The main tests that will be performed on the JaMoPP project will be done using a black box testing technique known as system testing. Because of the black box nature of System testing the focus will be on inputting source code and then gaining results from the output. A system test for this paper will consist of grabbing a large set of input source code and then using the Junit testing framework with a code coverage tool to ascertained results. The two tests that will be run on an input files will be:

1. A test to see if an input file can be parsed to create an instance of the Java meta-model.

2. After this a test will follow to take the newly created instance of the meta-model and then using the Java printer, return the model back to a text based representation. This text based instance will then be checked to see if it is the same code as the original file.

Following this a code coverage tool will measure the paths that were taken through the grammar and it will also measure the paths taken through the abstract syntax tree. These results can then be compared and also the code coverage will show how much of the system is left untested. Results will be gathered for a number of system tests to see if there is a correlation between the grammar and the abstract syntax tree or if it is possible to fully reach 100% coverage of both.

As mentioned in the last paragraph, the number of input files per round of testing will have to be of a large size and cover as much as possible of the Java language to fully test JaMoPP abilities. Some examples that could be used during this process would be to recreate some of the JaMoPP developers' tests and get code coverage readings from these. The tests that they used included Eclipse and Netbeans which are of a massive size and because of this they should bring in high coverage results. Another testing idea could be the use of beginner software development books and their example source code. These books teach students Java programming basics in a systematic and direct way, which would hopefully show the student most aspects of the language. For this reason they would make an ideal test for the JaMoPP system and should thus provide good coverage results.

The full overview of the testing process will now be discussed. A valid Java source code file is given to the JaMoPP system for parsing. This stage is a twofold process. The file is processed by the JaMoPP parser and as mentioned previously a model abstract syntax tree with unresolved cross references is created. Also at the same time the JDT parses the file and creates a standard Java abstract syntax tree. In the Junit test framework if this test is successful a green tick will appear beside the test name in the Eclipse GUI. If the parsing of the file is not successful or some problems are identify during the cross-referencing process the JaMoPP developers specified the system to return a parsing exception or run forever. The run forever case will be caught by a timeout exception in the Junit test framework. Any of the above exceptions will result in a failure in the Junit framework. A blue "X" will be placed beside the test name in the Eclipse GUI. Any problems with the Java heap size or

problems reading a file will register as an error in the testing framework and the test will receive a red "X" beside its name.

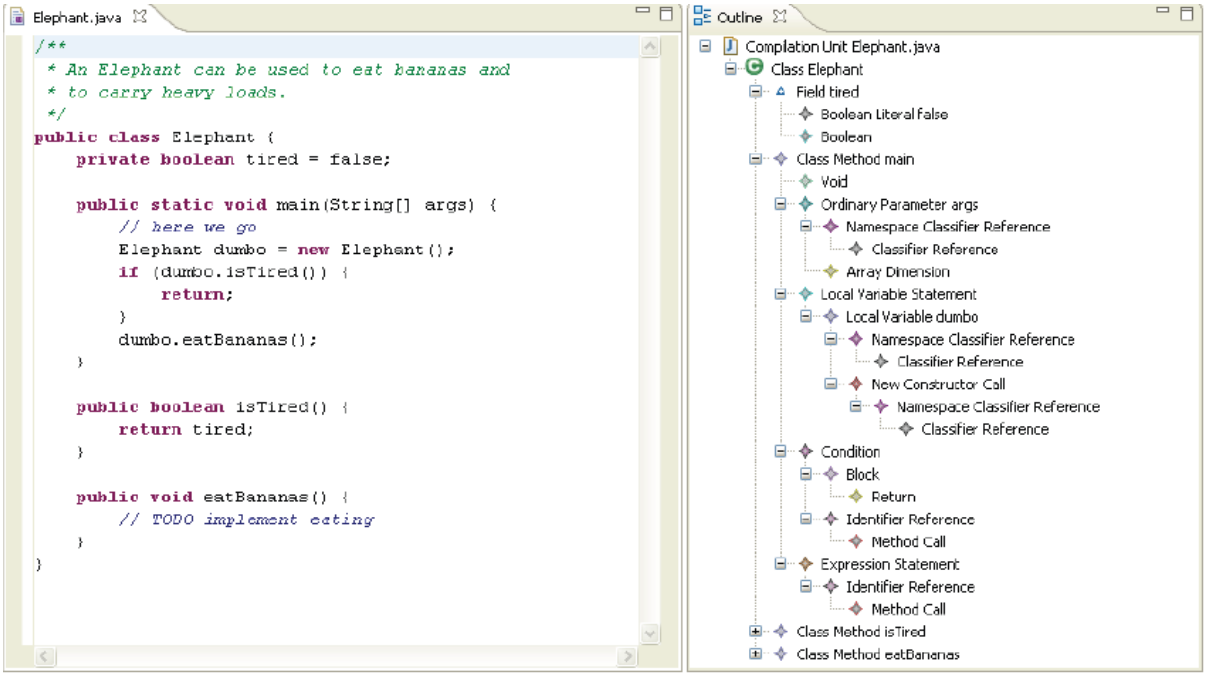

Figure 7: A Java class beside its Java meta-model instance which has the structure of an Ecore model [4]

Next the instance of the meta-model with the cross references is resolved and transformed back into text based code. This file is then taken and parsed using the JDT and another abstract syntax tree is created for this file. The second JDT abstract syntax tree is then compared to the original JDT abstract syntax tree using the JDT'S own abstract syntax tree matcher [1]. If these two abstract syntax trees are the same then the test passes.

Following these two tests the code coverage tool returns coverage measurements on all the files that were used during testing. The results for the Java parser and the abstract syntax tree will then be collected and compared. The coverage tool also colours the source code to show which lines of code have been used and which have not. In the case, where for example, a statement's true path has only been used, the tool will colour this statement a different colour from the case where both a statement's paths have all been traversed. More specialised testing could be created to achieve 100% coverage of these types of statements.

Section 4: Testing and Coverage Results

The JaMoPP project was heavily tested by its developers using open source industry sized software. They tested it against very popular Java development environments, web frameworks and some code generators. They did these tests, firstly to see if JaMoPP could be used in real world modelling development scenarios while at the same time testing JaMoPP's ability to handle as much of the Java 5 language as could be thrown at it. This paper will also need to conduct large scale tests on the JaMoPP project to test the system for as much of the Java language as is possible and then from these tests gain coverage information to analysis the parser and abstract syntax tree.

The first test that will be preformed will help to establish some baseline coverage readings and this will also help to set up a comparison between the coverage of the abstract syntax tree and the parser later. The source code that will be under test will be the classic HelloWorld Java program. This is usually the first program that is shown to new software developers and while it is easy to understand it uses a number of elements of the Java language. Looking at this first will allow for a more in-depth look at the underlying coverage results for the parser and abstract syntax tree. It will allow for the results to be built up from a simple well known program all the way to the massive scale tests required to cover as much of Java as possible. Later these tests will consist of many hundreds of files of source code where it will be difficult to establish which line of code maps to an execution of a certain path through the grammar or abstract syntax tree. As has been described in section 3, prior to getting the coverage results, the parse and reprint tests must be performed and then from the execution of these tests, coverage readings can be gained from the coverage tool about the percentage of the parser grammar and abstract syntax tree that have been executed.

Test Name	No Tests	Passed	Failed	Errors	Parser	AST
HelloWorld	2	2	0	0	25%	31.7%

Table 1: Shows the results of the HelloWorld tests

In the table above, the results from the HelloWorld tests are shown. It can be seen that on the Java input code, two tests have been performed successfully. In the first test the JaMoPP project was able to take the source code and create an instance of the Java meta-model. Then in the second test the JaMoPP project was able to take the model and transfer it back to text based source code which was then compared to the original and found to be the same. In the last two columns of the table, the coverage results for the parser and the abstract syntax tree are displayed. A closer look will now be taken at the coverage results for the two

components in order to show how these overall results are achieved and also to highlight some issues that have been seen during the experimentations regarding the layout of the packages that make up the abstract syntax tree compared to the one file of the parser.

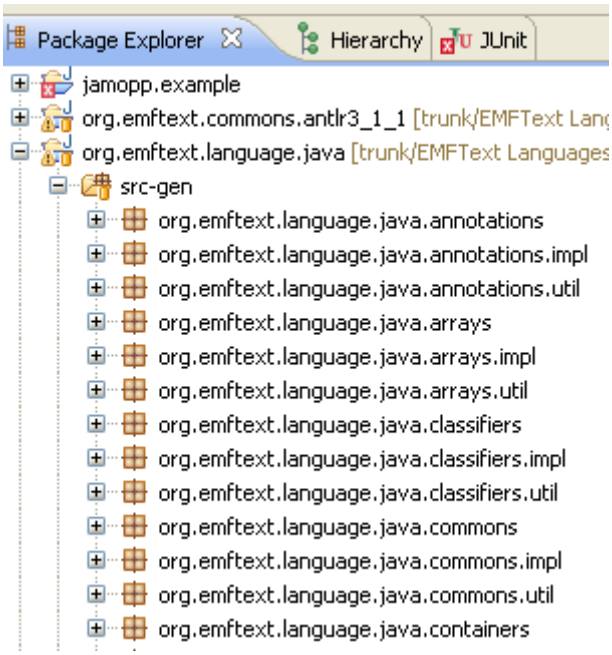

Figure 8: Partial view of the subfolders of the generated folder found in the org.emftext.language.java package.

The package org.emftext.language.java and its generated subfolder were described in section 3. It is this generated subfolder which contributes to the coverage readings for the abstract syntax tree. In table 1 we can see that 31.7% of the code of this subfolder was executed. Within this generated folder are many subfolders which are used to create the JaMoPP abstract syntax tree for the Java meta-model. As can be seen in figure 8 above, each subfolder of the generated folder corresponds to a component of the Java language; for example the arrays subfolders deal with every aspect of arrays for the Java language. As figure 8 shows there are 3 folders that hold the word array within their name and this is the same for all the components of the JaMoPP abstract syntax tree. For the array component these folders are called arrays, arrays.impl and arrays.util. The subfolder arrays holds all the interface classes that models each element of the array component of the Java language that helps to build the JaMoPP abstract syntax tree for input source code. The arrays.impl subfolder holds the implementing classes for the interface classes mentioned above, and this subfolder makes up all the contribution for the array component's coverage results. The final subfolder, arrays.util, is just some adapter classes for arrays. This folder returns a coverage result of 0% for all the tests that are to be discussed in this section.

As mentioned the Java parser is a single class that has methods which correspond to classes in each of the impl subfolders mentioned above. These classes and methods can then be compared side by side to contrast the parser and abstract syntax tree results. In figure 9 below, it shows the breakdown of the results for each of the Java components' coverage results for both the abstract syntax tree and the parser. Coverage results for each component of the abstract syntax tree are taken from the impl subfolders. While in the parser, for example any methods that deals with the array component classes, are added up and the percentage that this represents of the total possible array component is found.

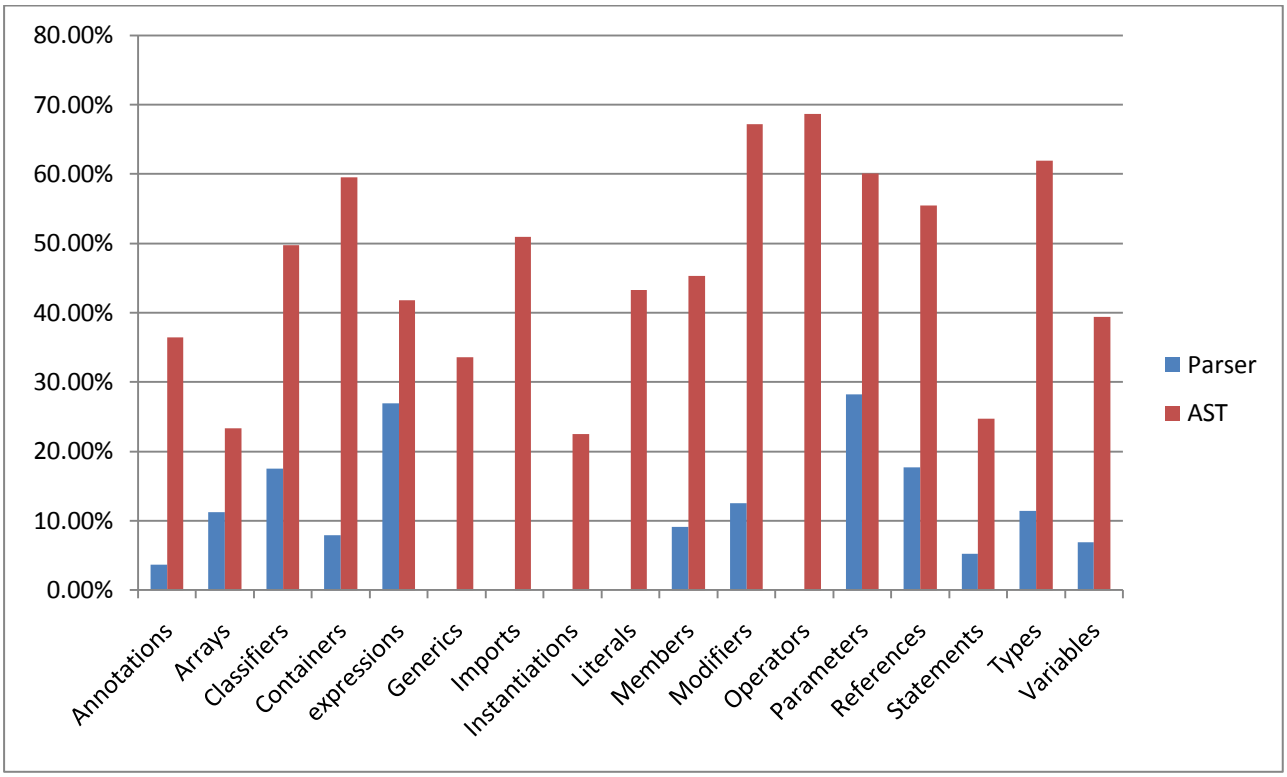

Figure 9: Break down of the Coverage results for the Parser methods that map to the corresponding impl folders of the org.emftext.language.java package. The names of the folders correspond to elements of the Java language.

As can be seen in figure 9, even though the HelloWorld input file contained no generics, imports and other such components of the Java language the abstract syntax tree that was created from the input file caused some of the generated folder's code to be executed. While in the parser, as is shown in the graph, no paths for generics or imports were traversed. The same is true for the other components of the Java language within the test. Some of the components had 0% coverage in the parser or significantly less than the abstract syntax tree. Let's take a closer look at one of the components of the Java language and compare the results of the parser and the abstract syntax tree.

The modifier component of the Java language has keywords that help set the visibility and accessibility of a class, its member variables, and methods [12]. These keywords would include public, private, protected, native and final etc. In figure 10 it can be seen that the abstract syntax tree has 100% coverage for most elements of this component including private, protected even though the only modifiers used in the HelloWorld class are public and static. In all the tests that were run during the testing phrase, even though in the abstract syntax tree all the modifiers got 100% coverage, the highest the results ever got within the parser is 62.4% which is the same as the result achieved by the public keyword in this test.

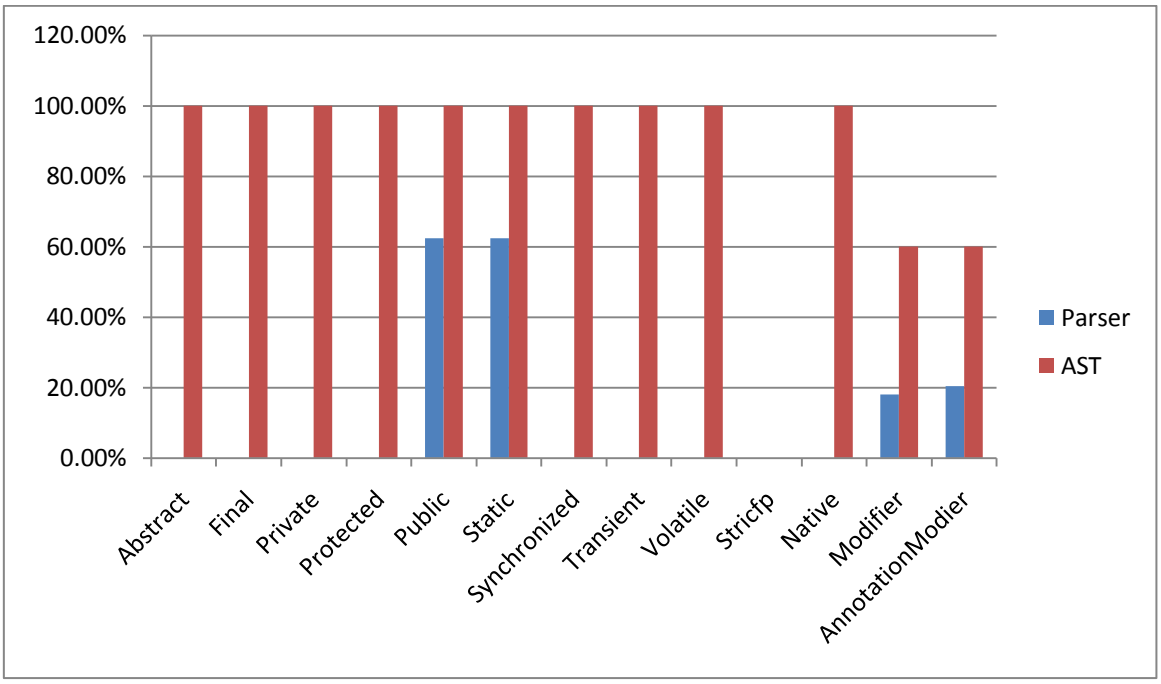

Figure 10: Graph of the coverage results for the parser and AST for the modifier component of the Java language.

The next round of tests will focus on testing the JaMoPP project using source code from software development books. The idea behind these tests is that these books teach Java from the ground up and would cover all the basics of the Java language. This would include the String class, loops and programming control flow, generics and collections and modifiers etc.

Test Name	No Tests	Passed	Failed	Errors	Parser	AST
HeadFirst	116	98	18	0	45.3%	40.6%
Sams	212	196	16	0	46.8%	41.7%
Game	374	222	152	0	51.3%	42.7%
Nutshell	78	63	15	0	47.4%	41.1%

All Tests	778	577	201	0	55.1%	44.5%

Table 2: Shows the results achieved from the software programming books. The All Test tests refer to all the tests from the books run together.

The table above shows the results from the software book tests. The HeadFirst [13] tests refer to the source code that is supplied with a book called 'Head First Java' released by O'Reilly Media. This book seems to have a focus on Java desktop application development. The Sams [14] tests above refer to the source code supplied with the book 'Sams Teach Yourself Java 6 in 21 Days'. This book, like the last, focuses on creating Java desktop applications but it does this using version 1.6 of the Java language. It was hard to find good source code on the web so this was a way of testing the language with expected failures. All the failures that were found during testing JaMoPP for this book were just problems with JaMoPP understanding Java 6 specifics. The tests named game [15] refers to the source code found in the book known as 'Developing Games in Java'. This book focuses on building games through Java. It was thought that this book would provide a good way to test the JaMoPP system for tread handling and maths operations. The failures that were found here are related to external libraries such as the javax library that doesn't seem to be able to be handled by JaMoPP. The nutshell [16] test refers to a book known as 'Java In A Nutshell'. This book focuses on building websites and website Java applications. The failures that were found here also relate to the javax library and GUI building errors. The All Tests test refers to all the tests that were run in the books that were discussed above but now these tests are run altogether to get the overall coverage results for this set of tests.

In the tests that where run in the table above, there were a lot of failures in the reprinting tests. The tests could all be parsed fine but when they where reprinted and compared to the original they were found to be different. The only time that the parse test can be made to fail is if it is given an invalid Java code file or if it times out within the Junit testing framework. Let's look at the failures that were found in the headfirst test cases. These should have all passed because they used the Java 5 language in a way that would be expected in any standard programming practice.

The tests seem to fail for a number of ways that the Java keyword import is being used within the headfirst tests. There were 18 failures out of 116 tests. Four of the tests fail because they import the javax class. The rest of the tests fail because they import classes that are part of the headfirst source code. The JaMoPP project does not seem to be able to import user created classes and this has many knock-on effects within the methods that are invoked from the imported class by the importing class. So JaMoPP can only import code that has

been particularly defined by the developers within the process of creating the textual syntax. JaMoPP also can't handle cases were a class creates a new Object instance of another user created class that is within the same folder as the class. So for example, in a folder there existed two classes, class1 and class2. Now within Class2 this statement appears, *Class1 example = new Class1();* this would be valid code within a Java file but yet JaMoPP is not able to handle this.

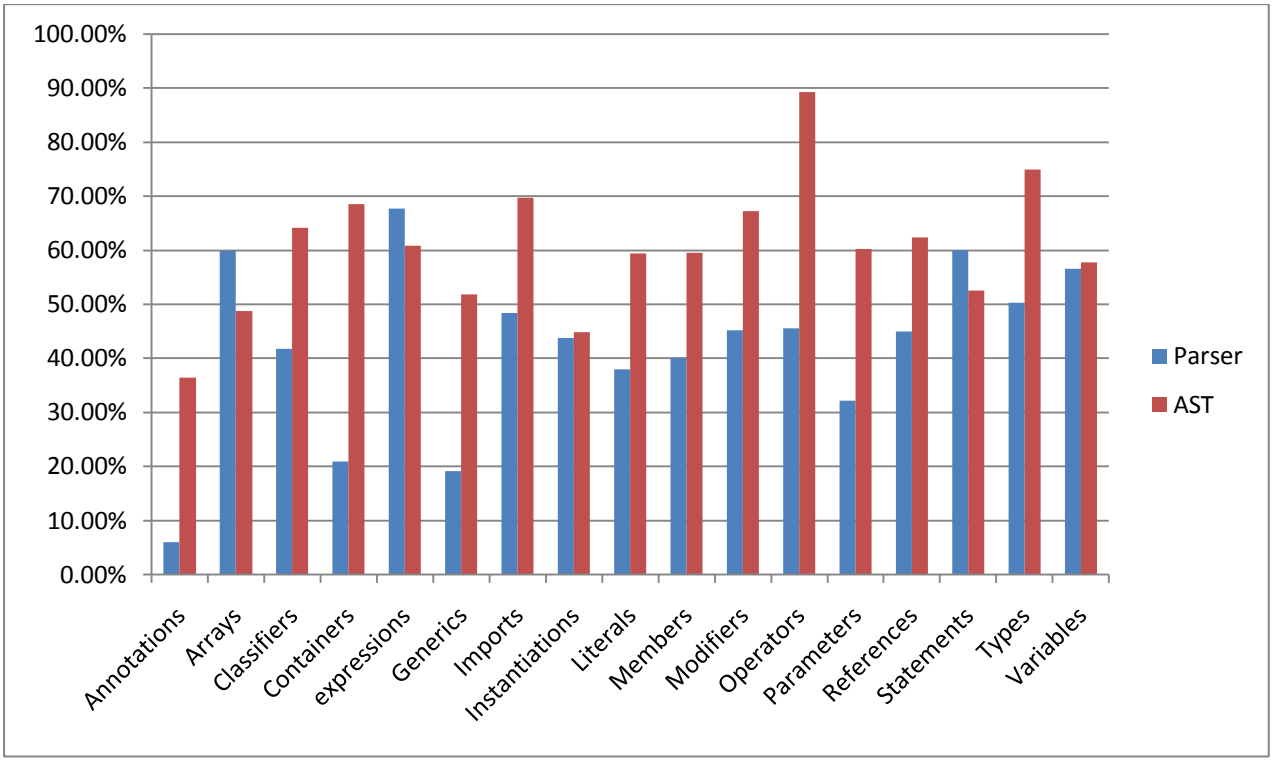

Figure 11: Coverage results for the All Test tests for the parser and the AST.

The results in figure 11 for the All Test tests results are not that much better than the results that were achieved in the HelloWorld tests. The abstract syntax tree results barely gained any more % coverage in any component while the parser did much better per component in the tests. It was decide to input a blank source code file into the JaMoPP project and see what abstract syntax tree results could be achieved. This would show how much of the abstract syntax tree code is executed regardless of the code in the input files. In figure 12 we can see the results of this test and the abstract syntax tree shows high coverage results per Java component even though the input file was blank. Only one method of the parser was executed. This method specifically deals with the instance of a blank model. The overall coverage for the parser from the blank test was 15.3% but when we look at the methods that are significant for this paper 0% is achieved except, in the case of the blank model method which got 21.1%, while in the abstract syntax tree results the overall is 24.6%. This may appear small but it must be remembered that the .util folders always achieve a 0%

so this brings the results down a bit but as can seen in the graph in figure 12, each of the .impl folders are quite high almost as much as during the book tests.

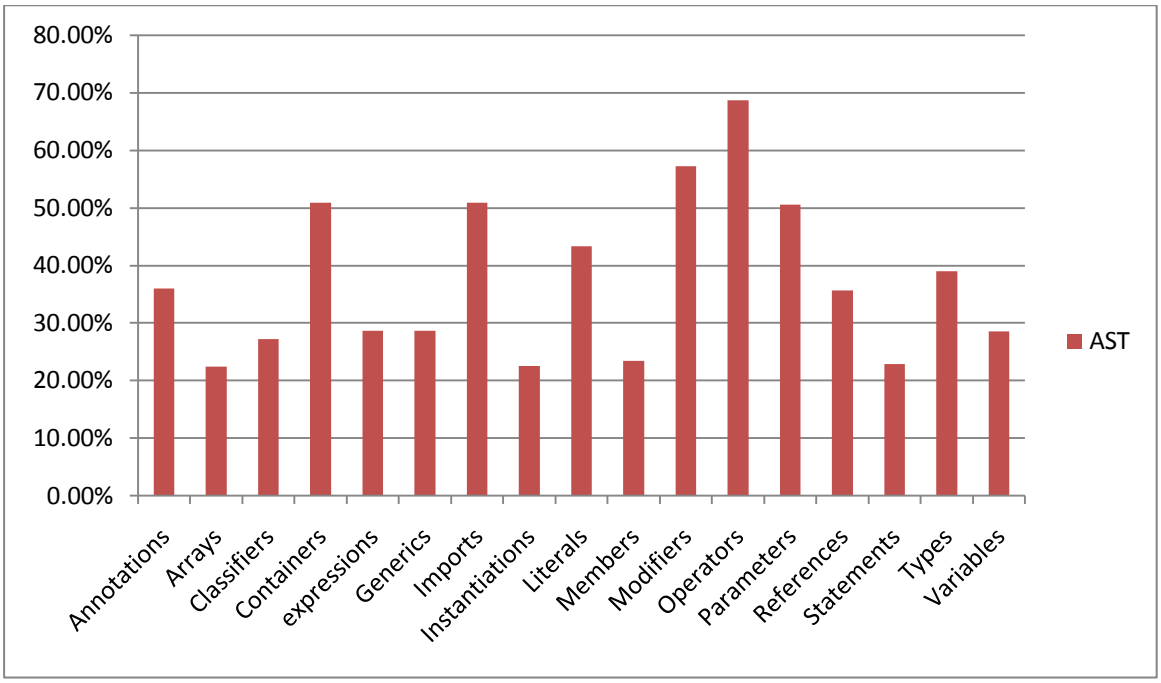

Figure 12: The breakdown of the AST results for a blank input file. The total AST result was 24.6%.

If a look is taken inside each of the .impl folders there is a file called (Java component)packageImpl file which scores almost 100% in each .impl folder and this appears to be by far the biggest file in any of these folders. For example in the modifier.impl folder there is a file called ModifiersPackageImpl this file achieves 92.9% coverage and the overall modifiers component result is 57.2%. The rest of the files that handle the actual modifier instances, for example public.java, achieve 0% coverage. These packageImpl files seem to handle creating instances of each class found within in the various .impl folders. It seems to create these instances whether they are used or not within the creation of a meta-model instance. So the question becomes, if the file with the biggest amount of code is getting almost 100% coverage and as has been shown in figure 10, most files within the .impl folder get 100% coverage in a simple program such as HelloWorld. Why can't a 100% result be got for the modifiers component with only a few additions to the HelloWorld file? Well the answer to this is that some files only ever reach a certain level of coverage within every test that was run. For example the second biggest class within this folder is called ModifiersFactoryImpl this class only ever gets as far as 52.1% coverage, the AnnotationInstanceOrModifierImpl class always gets 60% coverage and the ModifierImpl

class gets 60% coverage. These results occur regardless of the size and number of the input tests. These kinds of situations occur in every Java component of the abstract syntax tree.

Test Name	No Tests	Passed	Failed	Errors	Parser	AST
Andromeda 3.3	698	698	0	0	52.9%	43.9%
Apache-tomcat	1127	1127	0	0	60.4%	46.1%
Eclipse 3.4.1	16696	16690	0	6	62.9%	46.6%
Netbeans 6.5.1	31223	31167	53	3	68.1%	48.4%
JBoss 5.0.0	6414	6414	0	0	64%	47.4%

Table 3: The results achieved by the recreation of some of the JaMoPP developer's team's tests.

The tests shown in table 3 are the recreation of the tests performed by the JaMoPP developers when they were testing their system. These tests are thousands of files long and should cover as much of the Java language as is possible using a real world project. These tests should have all passed because the developers mention in their paper [JaMoPP] that any tests that they inputted passed. As can be seen in table 3 this is not true. The nine errors in the above tests occurred during the parse test due to a timeout within Junit. This could happen for two reasons:

1. The input file was so big that it took longer than the length of time specified in the Junit testing code.
2. The JaMoPP developers designed the parser to handle the case where it couldn't parse a valid input file, to either throw a parsing error or to run for ever. The run forever situation would then be caught by a timeout in the Junit testing framework.

There are 53 errors in the Netbeans test and these mostly occurred due to resolver issues in the reprint test.

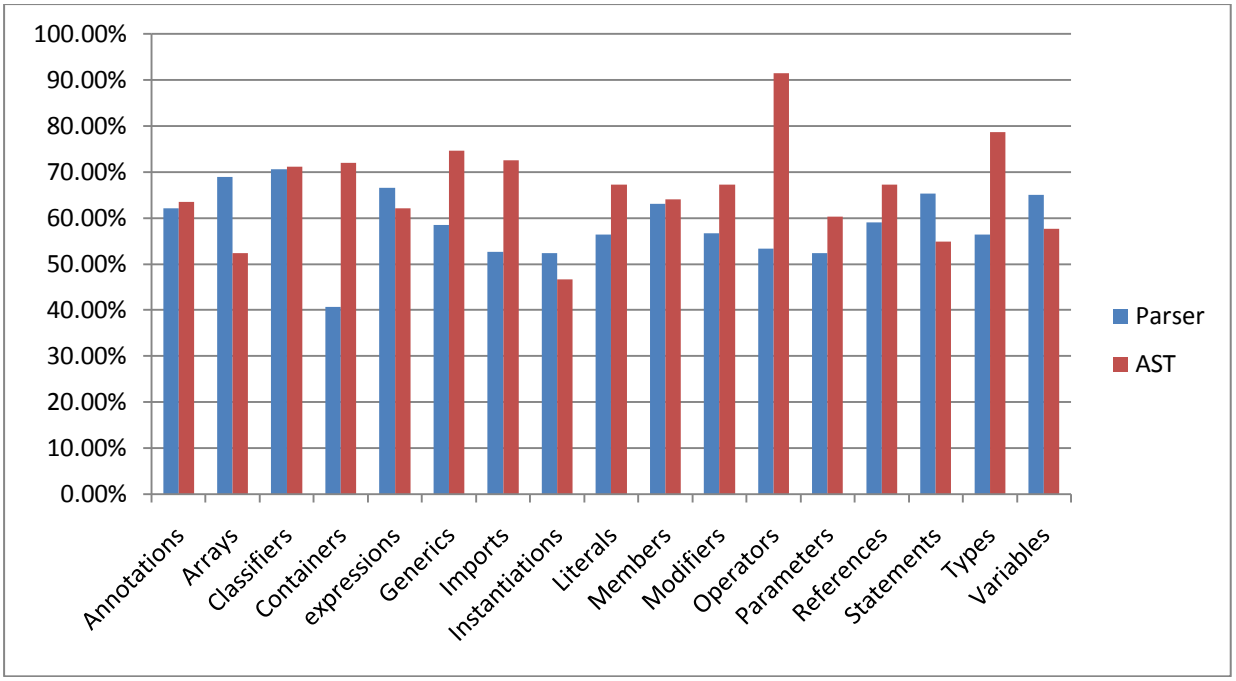

Figure 13: The breakdown of the parser and AST coverage results for the Netbeans test.

The above figure shows the breakdown of the parser vs. the abstract syntax tree coverage results for the Netbeans test. The parser results are getting closer and closer to full coverage as the amount of the Java language that is being used increases. But as has been mentioned a number of parser methods only ever reach a definite code coverage result no matter the amount of files that are inputted. For example in the modifier component the methods that deal with public, private, static etc only ever reach a result of 62.4% while in the abstract syntax tree 100% is achieved. If a look at the files and methods that make up the Java operators component results, the same situation can be seen. The operators component deals with addition, various kinds of assignment, plus plus and minus minus etc. These elements were found to achieve 100% coverage in the abstract syntax tree while only achieving 51.2% in the parser.

There doesn't seem to be a correlation between the results achieved by the parser and the abstract syntax tree. The abstract syntax tree results for most part float around the middle 40%'s. This is because certain files in each of the .impl folders only ever achieve a definite result. Also the second biggest file in each of the .impl folders, the factoryImpl class, only ever reaches a coverage result of around 50%. Each of the .util folders achieve a 0% coverage result for all the tests that were run and because of these factors the abstract syntax tree results don't vary much. The parser's coverage results can be seen to increase as the number of input files increase but a percentage of the parser's methods only ever reach a

certain result and never go higher. These results are different for each Java component so a trend cannot be ascertained from component to component.

Section 5: Conclusion

The JaMoPP project helps to bridge the gap between the two disciplines of coding and modelling. With this gap closing, this will help to bring the theory of MDSD methodologies into real word reality. The Eclipse project with its extendable architecture and open sourced nature contributed greatly to the construction of the project. The EMF plug-in for Eclipse provided the JaMoPP project with a well defined and well known meta-model to build their java meta-model on top of. They were the first developers to create a standardized java meta-model which can be manipulated on and be handled in the same manner as any other EMF model.

Since the developers goal would be to get the JaMoPP project out and be used by real world developers and researchers there is little public information about it on the web. The project itself comes in two options, one called the stable version and other one is the most up to date version that the developers themselves are using to fix problems and add features. It would be thought that any serious development or research work using the JaMoPP project would require the most up to date version to ascertain its current limitations and future possibilities. The most up to date version also contains the base testing packages that are required to ascertain the overall benefits and likely success using the system. Yet the only information on their website is for installing and using the stable version. At the time that the JaMoPP project was being installed for use with this paper the only way to get the full most up to date version was by contacting the developers directly. Since then the website has been updated with a full repository of code and the very important technical document is also now available. Also on the website the grammar and textual syntax is up and can be seen.

On the website there is a link to a Java API for the generated subfolder of the org.emftext.language.java.resource.java package. This is the folder that contains the generated ANTLR files such as the parser, lexer and token Streams etc. The EMF Framework hides the underlying implementation of the model resource factories, so it would never be required to access the classes of the generated folder directly. At the beginning of the testing phase for this paper a false impression that the parser needed to be accessed

directly was held until the JaMoPP developers were contacted directly and it was realised that the EMF framework handles this. It cannot be seen why the API would be up when it just leads users to confusion.

The tests conducted by this paper reviled a number of things about the JaMoPP project. Firstly most of the files that were inputted into the system as tests passed. Most of the failures in the 'book' set of tests, failed due to what appears to be various issues with importing user created packages and invoking methods from user created classes. The JaMoPP developers claimed that any files that they used for their testing purposes passed but upon recreation of some of their tests it was found that some inputs failed. These failures mostly occurred in the Netbeans test due to resolving issues in the reprint tests. A number of errors were also found in the Netbeans and Eclipse tests. These errors occurred because of timeouts in the Junit framework.

After looking at the coverage results for both the parser and the abstract syntax tree there was no obvious overall correlations found. Some of the individual files of the abstract syntax tree got 100% coverage and within the corresponding methods of the parser it was found that the parser only reached a certain level of results. No matter how big the input file set became, the abstract syntax tree result comes in at around the high 40% mark. While the parser results increased as the amount of the Java language that was been used increased, a number of the methods that contributed to the final parser coverage result started to reach certain definite levels.

The system was tested against big industry sized software and small programs. It would be hard to tell how to attain a 100% coverage result in the parser. Within the technical document the developers mention adding grammar rules to handle particular coding situations but don't really supply information about these scenarios. Maybe to get full coverage, these situations would need to be reproduced if they were not occurring during the tests. The abstract syntax tree had hit its coverage peak and most files were either at 100% coverage or stopping at a particular level. The parser was increasing as mentioned, as the amount of code increased but certain methods were starting to show a definite level of reoccurring results. Java also supplies many ways to accomplish the same feat for example if we take a look at the plus plus element of the java language. A variable can be incremented either by coding this ++variable or by this variable++. To get a 100% coverage of the plus plus element in the grammar both of these scenarios must occur in the input code. The same

would be true for other elements of the java language and also this would lead on to the issue of programming styles. For example Netbeans and Eclipse would be programmed in a certain programming style by its developers and this could miss out on some paths through the grammar for certain java elements. Also Java supports legacy ways of coding Java that may have fell out of practice by more modern programmers. All of these issues would lead to problems achieving a 100% result.

5.1 Future Work

The JaMoPP project opens up a number of areas in coding and modelling. The JaMoPP developers themselves have suggested some areas of future research including generating, analysing and visualising code. This year the Java 6 language was released. This means that the Java meta-model and textual syntax needs to be updated to reflect this. Other ideas for other areas of testing the JaMoPP project would be using it in a practical sense. Maybe inputting some test files and then using the generated meta-model instances for performing transformations between different EMF model types. Then these tests could be looked at for coverage results and see if the abstract syntax tree's .util folders are used during this process. This form of testing would also test to see how well the JaMoPP's Java meta-model operates and fits within the EMF Framework. Another testing area that was not covered in this paper was the extended parser that uses the BCEL project to read Java source code. Coverage results could be got for this and then also be compared to the abstract syntax tree. Maybe models generated from the BCEL code could also be tested to see how well they get on in EMF model to EMF model transformations.

References

[1] Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende. *JaMoPP: The Java Model Parser and Printer Technical Report.* TUD-FI09-10 September 2009. Institute for Software and Multimedia, Technical University Dresden, Germany.

[2] Mark Hennessy and James F. Power. *Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software.* Empirical Software Engineering, Vol. 13, No. 4, August, 2008, pp. 343-368.

[3] Frank Budinsky, David Steinberg, Ed Marks, Raymond Ellersick and Timothy Grose. *Eclipse Modelling Framework: A Developer's Guide.* Published by Addison Wesley, August 11, 2003. ISBN number: 0-13-0142542-0.

[4] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert and Christian Wende. *EMFText and JaMoPP - Tool Presentation.* Institute for Software and Multimedia, Technical University Dresden, Germany. Published in 2009.

[5] Peter Graff. *Xtext vs. EMFText: Development Process.* Published on November 23rd 2009. http://pettergraff.blogspot.com/2009/11/xtext-vs-emftext-development-process.html. Last accessed 26th of January 2010.

[6] Terence Parr. Information on ANTLR software: http://www.antlr.org/. Last accessed 26th of January 2010.

[7] Jendrik Johannes. *EMFText overview.* Published on 13th February 2009. http://st.inf.tu-dresden.de/reuseware/index.php/EMFText_Overview. Last accessed 26th of January 2010.

[8] BCEL software information: http://jakarta.apache.org/bcel/. Published on 3rd June 2006. Last accessed 26th of January 2010.

[9] Jendrik Johannes. JaMoPP project home page: http://jamopp.inf.tu-dresden.de/. Published on 5th of March. Last accessed 12th of January 2010.

[10] Jendrik Johannes. *Video demonstration of installing the JaMoPP project's stable version*: http://www.emftext.org/index.php/EMFText_Installation_Screencast. Published on 5th of March. Last accessed 26th of January 2010.

[11] Junit software information: http://www.junit.org/. Last accessed 26th of January 2010.

[12] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *Java Language Specification, 3rd Edition.* Published by Addison Wesley, June 24, 2005. ISBN number: 0321246780.

[13] Kathy Sierra and Bert Bates. *Head First Java, 2nd Edition.* Published by O'Reilly Media, May 11, 2009. ISBN number: 0596009208.

[14] Rogers Cadenhead and Laura Lemay. *Sams Teach Yourself Java 6 in 21 Days, 5th Edition.* Published by Sams, June 4, 2007. ISBN number: 0672329433.

[15] David Brackeen. *Developing Games in Java*. Published by New Riders Games, August 31, 2003. ISBN number: 1592730051

[16] David Flanagan. *Java In A Nutshell, 5th Edition.* Published by O'Reilly Media, March 15 2005. ISBN number: 0596007736.

[17] Frank Budinsky, David Steinberg, Marcelo Paternostro and Ed Marks. *EMF: Eclipse Modelling Framework, Second Edition.* Published by Addison Wesley Professional, December 16, 2008. ISBN number 0-321-33188-5.

[18] Javac software information: http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javac.html. Last accessed 26th of January 2010.