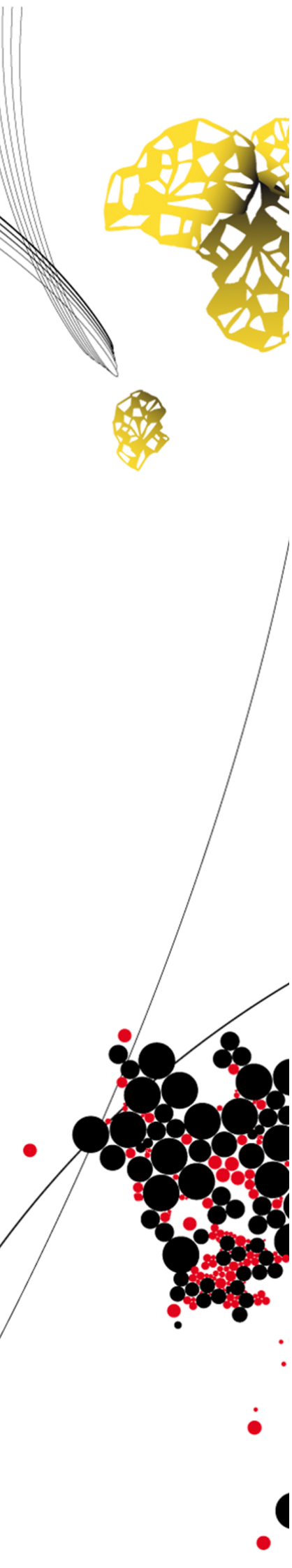


UNIVERSITY OF TWENTE.

Computer Science

Code instrumentation with Mod-BEAM

Roman Rinke Armando Wiedijk

MSc. Thesis

July 18, 2019

External Supervisor:
prof.dr. C. Bockisch

Faculty of Mathematics
and Computer Science
Philipps University of Marburg
35037 Marburg
Germany

Supervisors:
dr. A. Fehnker
prof.dr.ir. M. Aksit

Formal Methods and Tools
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

There are various methods and techniques for a developer to keep their code readable and reusable. Design patterns have become important in modern development in order to realise this. There exists code, however, that you would not want in production. An example of this would be code that is generated by instrumentation tools. Tools might want to generate code to help create performance benchmarks, or to analyse the code-coverage of your test-cases. ”Code instrumentation” is a term commonly used to describe the task of adding such generated code. What we focus on in this thesis is code-instrumentation for Java projects.

When a Java file is compiled, the compiler outputs a .class file with the low-level instructions that the Java Virtual Machine can execute. Java code instrumentation can be performed on the actual source-code (before compiling) or on the bytecode itself (after compiling). Multiple frameworks that allow for Java bytecode instrumentation. Some of these frameworks are a bit dated, but they all share the common interest to allow for easy, fast and maintainable bytecode manipulation.

From the hypothesis that existing tools that make use of bytecode instrumentation have potential overlap, the tool ”Mod-BEAM” is introduced. Mod-BEAM is a new framework that makes use of model-driven engineering to convert any Java class into a clean, modifiable model representation of the original bytecode. Mod-BEAM stands for ”Modular Bytecode Engineering and Analysis with Models”.

When working with model-driven engineering, there exists the concept of a model and a meta-model. In the case of Mod-BEAM, the meta-model is a representation of all possible Java code (from classes to instructions). The model would be an actual class with fields, methods, instructions, etc. What Mod-BEAM does for us is to transform the original Java .class file (containing the bytecode for the compiler) to the corresponding model that inherits the Java metamodel as defined by Mod-BEAM. This newly generated model can then be transformed using a reusable template through the use of a model-transformation language. This resulting model can then, through Mod-BEAM, be converted back into an executable Java .class file. Summarised, this means that we go from .class to .jbc (bytecode model), then from .jbc to a .jbc with additional instructions. Finally, we go from the modified .jbc back to a .class file, which now has all the modified data in it.

As part of the problem analysis, we identified tools that perform bytecode instrumentation. We proceed to identify which tools have a potential overlap in their ”primitive concepts”. Primitive concepts would be concepts such as finding key-points, instrumenting touch-points or inserting probe-points. Along with the identification of these concepts, we look at how they are implemented and which concepts overlap. From this, three tools came into focus. Those three tools are Cobertura, JaCoCo and AspectJ. The first two perform Code-coverage Analysis, while AspectJ is a tool used for Aspect-oriented Programming. By re-implementing the concepts found in these tools through a model-transformation language in combination with Mod-BEAM, we aim to identify the advantages and disadvantages of this new approach through model-driven engineering.

The instrumentation functionality of Cobertura is the first to be re-implemented. The original code is analysed to determine how it works and what the statistics such as the complexity are. The code in our model transformation is proven to have most of the functionality for class instrumentation, producing near-identical results to Cobertura while being far less complex. As the code is concentrated in a single file, it comes at the cost of class quality, however.

The second tool, JaCoCo, was due to the straightforward nature of the original implementation even easier to re-implement. Similar to Cobertura, results were analysed and the results were more compact and less complex, facilitating higher usability.

The third and last tool, AspectJ, was after some more in-depth research found to have concepts that differed more than expected from the concepts of the Code-coverage Analysis tools. We have provided an analysis of the workings and a discussion on how a model-driven implementation of the weaving process could be created.

The unique contributions of this thesis are: A reimplementation of the core instrumentation functionality of Cobertura and JaCoCo, a comparison between our implementation as model-transformation and the original implementations using Mod-BEAM, a look at the reusability between our implementations and a discussion on a hypothetical reimplementation of the weaving functionality of AspectJ.

Preface

This document is the Master Thesis ”Code Instrumentation with Mod-BEAM”. This document is effectively a thesis continuation of a smaller research, ”A Study on Code Instrumentation Approaches”, performed in the months prior to the creation of this thesis. This thesis has been written to fulfill the graduation criteria of the Computer Science - Software Technology programme at the University of Twente.

I would like to thank Christoph for the help he presented when using Mod-BEAM and the speedy fixes to any issues I encountered. I would also like to thank Ansgar for the clear feedback on the Thesis, and Mehmet for keeping me focused throughout the writing process. Lastly, I would like to thank my family for standing with me and for reminding me not to worry too much. Working with bytecode and the various tools that modify it has greatly deepened my knowledge of what happens under the hood of Java.

Contents

List of acronyms

AOP	Aspect Oriented Programming
CCA	Code-coverage analysis
CYC	Cyclomatic complexity
ETL	Epsilon Transformation Language
LCOM4	Lack of Cohesion Method version 4
LoC	Lines of Code
MDE	Model-Driven Engineering
NCLoC	Non-Commenting Lines of Code
QVTo	Query/View/Transformation Operational

Chapter 1

Introduction

This chapter contains the necessary background information, the research goal of this thesis and how the thesis is organized.

1.1 Background

There are various methods and techniques for a developer to keep their code readable and reusable. Design patterns have become important in modern development in order to realise this. There exists code, however, that you would not want in production. An example of this would be code that is generated by instrumentation tools. Tools might want to generate code to help create performance benchmarks or to analyse the code-coverage of your test-cases. ”Code instrumentation” is a term commonly used to describe the task of adding such generated code. What we focus on in this thesis is code-instrumentation for Java projects.

When a Java file is compiled, the compiler outputs a .class file with the low-level instructions that the Java Virtual Machine can execute. Java code instrumentation can be performed on the actual source-code (before compiling) or on the bytecode itself (after compiling). Multiple frameworks that allow for Java bytecode instrumentation. Some of these frameworks are a bit dated, but they all share the common interest to allow for easy, fast and maintainable bytecode manipulation.

In Part I of our research we identify which tools perform bytecode instrumentation and which instrumentation libraries are used for these tools. We also identify which concepts are implemented by these tools and how they are implemented. We do this in order to prepare for a larger research, namely Part II which aims to validate the usability and reusability of a new bytecode instrumentation tool that works through model-driven engineering.

When working with model-driven engineering, there exists the concept of a model and a meta-model. In the case of Mod-BEAM, the meta-model is a representation

of all possible Java code (from classes to instructions). The model would be an actual class with fields, methods, instructions, etc. What Mod-BEAM does for us is to transform the original Java .class file (containing the bytecode for the compiler) to the corresponding model that inherits the Java metamodel as defined by Mod-BEAM. This newly generated model can then be transformed using a reusable template through the use of a model-transformation language. This resulting model can then, through Mod-BEAM, be converted back into an executable Java .class file. Summarised, this means that we go from .class to .jbc (bytecode model), then from .jbc to a .jbc with additional instructions. Finally, we go from the modified .jbc back to a .class file, which now has all the modified data in it.

Model-Driven Engineering (MDE) has the advantage that through a single metamodel, any model that is represented by that metamodel can be transformed or reused. This means that in our case, any Java class can be transformed into a modified class through the use of a model-transformation. For a developer, this would mean that model-transformation techniques such as mappings can be used, where, for example, a code instruction of a specific type could always generate a set of inserted followup instructions in a consistent pattern. Mod-BEAM can be seen as complete back-end functionality, while a model-transformation has the concrete implementation. Developers would not have to deal with issues that involve the back-end, and should, in theory, be able to focus purely on creating the desired result.

We hypothesize that through the use of Mod-BEAM, we can improve the reusability or potentially simplify the original implementation of code-instrumentation for a tool. This is because a model-transformation grants more freedom in what we can do while working with a metamodel that is easier to understand than plain bytecode.

1.2 Research Goal

The objective of this assignment is to perform in-depth research on some of the approaches identified in Part I. This includes an overview of the complexity of the investigated concepts, an overview of the redundancy in the various concepts and how they overlap, and a possible solution to exploit reuse to reduce redundancy using Mod-BEAM.

The research will center around the creation of a library of model transformations that makes use of Mod-BEAM that reimplements some of the shared concepts between investigated tools. This tool should serve as a reusable base and will be compared with the original implementations using a set of criteria, such as the reusability of the implemented concepts in the tool, how usable the actual tool is in terms of compile-time or run-time instrumentation and how hard it is to maintain and

modify the tool.

The unique contributions of this thesis are: A re-implementation of the core instrumentation functionality of Cobertura and JaCoCo, a comparison between our reimplementation as model-transformation and the original implementations, a look at the reusability between our implementations and a discussion on a hypothetical implementation of the weaving functionality of AspectJ.

1.3 Report organization

The contents of the report are organized as follows. Part I contains the research that was performed as part of Research Topics. This report serves as a base from which we derive what we can possibly create in Part II. Chapter 2 describes which requirements and research questions are answered in this report. Chapter 3 contains a set of criteria on how to measure our implementation, as well as a small study on the advantages and disadvantages of various model-transformation languages. It also provides an analysis of valid entry points for the implementation-phase of the research. Chapter 4 contains information about the implementation, measurement, test-data and reimplementation of the first tool, namely Cobertura. Chapter 5 proceeds by describing the data from the second implementation and reimplementation of JaCoCo, along with details on the newly added reusability between the aforementioned tools. Chapter 6 then explains a hypothetical implementation of AspectJ. Finally, Chapter 7 provides a conclusion, discussion and recommendation for future work.

Part I

Problem Analysis

Research Topics

A Study on Code Instrumentation Approaches

Research Topics | *Final*

Roman Wiedijk

University of Twente

July 18, 2019

Abstract

With an ever-increasing amount of development tools available for a Java developer, it has become unclear just where and how code-instrumentation is being used. In this research we identify the various approaches for which code-instrumentation is being performed and we analyse how code-instrumentation has been implemented in various tools and frameworks. We proceed by comparing various important, primitive concepts that these tools and frameworks share, and we investigate the viability of an alternative framework that makes use of MDE in order to implement code-instrumentation.

Contents

1 Introduction

This is the document for the course Research Topics of the University of Twente. This document contains a research on code instrumentation. The goal of this research is to identify code-instrumentation approaches, to identify actual implementations and to investigate overlap between these implementations. More information about the motivation of this project can be found in section 2. The report starts by explaining the motivation for this research in section 2, continued by a listing of the necessary requirements (what data will need to be gathered) and the research questions (which concrete questions can give results that fulfil all these requirements) in section 3. Section 4 then explains the methodology in which this research was performed. Section 5 give an overview of the results that were created from this research. Section 6 draws a conclusion from the aforementioned results, and section 7 explains how this research can be continued.

2 Motivation

Java is a popular programming language. Through the usage of JavaC, the Java compiler, Java code can be transformed into native bytecode. This bytecode can then be interpreted and run on the Java Virtual Machine (JVM). While programs grow more complex, people want to make their lives easier by adding more automated functionality to their programs without modifying the original source-code. Such actions might include profiling or the extension of code without modifying the original classes.

Any form of applying metrics or additional information to code will require some form of code instrumentation. Code instrumentation could be used in order to gather analytical data, such as performance metrics (profiling). Profiling is something that is used in a lot of projects nowadays and is something that can be considered mandatory for some projects. Profiling is often implemented by adding instructions to the start and end of methods. As such, there are various forms of profiling and also various implementations on the market. There also exist different ways in which code instrumentation can be performed, such as source instrumentation where the source code is directly instrumented, or byte-code instrumentation for which the generated .class files are instrumented with additional code. Furthermore, code instrumentation can be performed either in a static manner or in a dynamic manner. For static code instrumentation, the code or class files are updated directly and stored on disk, while for dynamic code instrumentation the changes are applied when the files are loaded by the JVM.

While code instrumentation is quite simply the supplementing of the native bytecode with additional instrumentations, code transformation is to allow the (source-)code to be transformed into something else entirely. An example of this would be the transformation of source-code into Abstract Syntax Trees (AST). An AST might be easier to work with or to maintain, but might also be more easy to interpret for tools that focus on analysis or code-coverage. An example would be how Spoon[1] allows for its own form of transforming and analysing Java bytecode through the usage of “processors“, an object type that is a combination of query and analysis code. Code transformation is an example of an imperative way of transforming the original code, e.g. plain Java into another format. Imperative means that the used code knows about the details of the implementation, as opposed to a declarative mark-up.

A more complex action that can be performed on the source code would be something akin to code extension. Code extension could be performed using, for example, Aspect Oriented Programming (AOP). AOP allows for the usage of "aspects" which stand separate from actual class objects. These aspects could allow for supplementing code with which you could implement programming concerns that would normally fit into multiple classes or packages, such as security concerns or logging functionality. These aspects can be "weaved" together into the bytecode of the application. Weaving would be another example of code-transformation.

Model-Driven Engineering (MDE) is a fairly recent software development methodology trend. Through the usage of a source metamodel (e.g. Java bytecode), a target metamodel (e.g. Java) and a transformation syntax (e.g. Xtext). The distinction between an imperative and declarative transformation becomes important at this point, as there exist transformation languages that support either of those or a mixture of both. Both approaches have their own advantages, with a declarative approach being more "strict" and often easier to analyze, while an imperative approach gives more control on the sequence of the performed transformation. Models that fulfil the specifications of the source metamodel can be transformed into a model that fulfils the specifications of the target metamodel. By making use of templates, re-usability of the transformation code can be facilitated.

[1] http://spoon.gforge.inria.fr/

2.1 Problem & Goal

The "problem" for this research is that we have a new framework to with, called Mod-BEAM. Mod-BEAM allows for code-instrumentation through the usage of MDE. A suspected issue with existing code-instrumentation implementations is the lack of reusability, and the high amount of implementations that are extremely similar to that in other tools. We want to see if this new tool is viable to be used in order to increase reusability in existing approaches, but there exists no clear overview of the current state of reusability for the tools that are already available. The goal of this research is to identify and compare the various implementations of code instrumentation and code transformation that make use of the Java Virtual Machine in order to create the aforementioned overview.

In order to do this, we initially have to determine for which purposes code-instrumentation is frequently used. Afterwards, we will need to define which tools make use of code instrumentation and which libraries are used (if any). For the actual implementations, we can determine how it works and which primitive concepts are defined. For each of these tools, we can determine which parts of these primitive concepts are being used. If no external library is used, then that means that the tool has its own implementation and thus its own set of primitive concepts.

Using this new-found information, we will be able to easily compare both the techniques used in these implementations (e.g. what pattern, which technologies) and which primitive concepts have been implemented. For each tool, we will be able to compare which specific concepts were required in order to get the tool to work. Using a proof-of-concept, we will then be able to identify if Mod-BEAM, the new model-driven approach, can be used to substitute for one of the existing implementations / libraries as a valid alternative. As this is only a precursor to a larger research, the focus will lie more on the identification and comparison, while the investigation on the suitability of Mod-BEAM will be kept as a simple Proof of Concept.

2.2 Quality considerations

As research is performed on the various tools that implement code-instrumentation, various quality aspects of the examined code can be taken into consideration. These quality aspects can serve to identify which approach and which tools are easier to read, and therefore also most likely easier to maintain. Quality aspects that are taken into consideration are: Lines of Code, Separation of Concerns / Modularity and the general readability of the code. Besides this, it is possible to identify whether the tool is able to perform the code-instrumentation at compile-time or during runtime (e.g. through code hotswapping).

3 Research Questions

This section separates the concerns of the research into a set of requirements in order to create a clear understanding of what needs to be done. These requirements should be an indicator for the research questions. The requirements are split into short-term requirements for this research (R1-R7), and long-term requirements for a more detailed research (R8-R13). These requirements are then converted into actual research questions. Research questions exist as questions that should cover all the requirements for the appropriate section. Long-term research questions are considered to be future work, and are therefore not covered.

3.1 Short-term

The requirements that can be determined for the short-term (this research) can be separated into a goal requirement (the end goal of this product) and product requirements (the specifics in order to reach this goal). As such, each product requirement is part of the goal requirement.

3.1.1 Goal

R1 The research should create an overview of existing code instrumentation approaches and how they conceptually work, describing overlap in the concepts of these approaches and comparing them to Mod-BEAM. As result, the research should give a clear understanding of the current state of code-instrumentation approaches.

3.1.2 Product

R2 The document should provide an overview of existing code instrumentation approaches

R3 The document should provide an overview of tools used for each code instrumentation approach

R4 The document should provide an overview of libraries used for each code instrumentation tool

R5 The document should provide an overview of the primitive concepts that are implemented by these tools & libraries

R6 The document should provide an overview of how these primitive concepts are implemented by these tools & libraries

R7 The document should identify conceptual overlap in these varying implementations

R8 The document should provide insight of how MDE can be used in terms of code-instrumentation

3.1.3 Questions

The short-term research question to be answered is:

- Which approaches currently exist for code-instrumentation and how do they compare? (R1).

The following sub-questions can be answered in order to answer the main research question:

- Which code-instrumentation approaches and implementations exist for the JVM? (R2, R3, R4)
- What technologies and primitive concepts are used in the aforementioned implementations? (R5, R6)
- Is there overlap between these implemented concepts? (R7)
- In which manner is model-driven engineering a suitable approach for code-instrumentation? (R8)

This research question will provide leeway to the long-term research question: In which way is the implementation of instrumentations as model transformation superior/inferior to the implementation as code transformation?

3.2 Long-term

3.2.1 Goal

R9 The research should create an overview of existing code instrumentation approaches and how they work in detail, describing overlap in the implementation of these approaches and researching the viability of Mod-BEAM. As result, this study should give a clear understanding on existing approaches and implementations, with insight on the advantages and disadvantages of each approach.

3.2.2 Product

R10 The document should provide an overview on the complexity of existing code instrumentation approaches

R11 The document should provide an overview on the redundancy within each code instrumentation approach

R12 The document should provide an overview on the redundancy compared to other code instrumentation approaches

R13 The document should provide possible solutions on resolving redundancy in code instrumentation approaches

R14 The document should identify if Mod-BEAM can be used to decrease redundancy in code instrumentation approaches

4 Research Method

In order to answer the questions mentioned in section 3, we defined the research method described in section 4.1. This section will explain how the research is performed and how the various identified techniques are compared with one another.

4.1 Methodology

The main question is "Which approaches currently exist for code-instrumentation and how do they compare?", and the various sub-questions are used in order to lead to an appropriate answer for this question.

The first sub-question: "Which code-instrumentation approaches and implementations exist for the JVM?" is used to determine the existing approaches that will need to be investigated. This will be done by reviewing common approaches and usages of code-instrumentation. The first question aims at identifying all code-instrumentation approaches in order to create a categorized overview of existing approaches, so that actual tools and libraries can be investigated in the second sub-question.

The second sub-question: "What technologies and primitive concepts are used in the aforementioned implementations?" aims at identifying usages (tools) and concrete implementations (libraries) in order to identify how the aforementioned approaches were implemented. This part of the research can be performed by looking up instrumentation libraries and checking which tools make use of these libraries, or by looking up tools that make use of some form of instrumentation and then confirming how this instrumentation was implemented. The actual implementation will be researched through the usage of available papers in combination with the original source-code of the tool or library.

The third sub-question: "Is there overlap between these implemented concepts?" aims at determining conceptual overlap between the various implementations and techniques. Important to note is that the question is about the conceptual implementation (e.g. technologies used and primitive concepts). This can be researched by mainly looking at the previously answered research questions. More information about the actual comparison can be found in section 4.2.

The fourth and final sub-question: "In which manner is model-driven engineering a suitable approach for code-instrumentation?" pertains to the suitability of model-driven engineering for code-instrumentation approaches. This will be researched by creating a proof-of-concept that will mimic a code-instrumentation approach or implementation using Mod-BEAM.

As a result of answering the various sub-questions, a conclusion can be formed. This conclusion will then be the answer to the main question of this research.

4.2 Comparison criteria

In order to compare the various implementations of code instrumentation approaches as required for the sub-question "Is there overlap between these implemented concepts" in section 3.1.3, a set of criteria is defined. These criteria can be performed for two separate instances, namely:

- Overlap in primitive concepts for the same code-instrumentation approach
- Overlap in techniques used for different usages of code-instrumentation (e.g. ASM visitor pattern for profiling compared to code-coverage)

For each of these items, the following points can be answered in order to deduce conceptual overlap:

- Is there overlap in techniques and concepts used?

In this manner, all necessary information can be gathered.

4.3 Proof of Concept

In order to prove that Mod-BEAM is viable for code instrumentation, a Proof of Concept (PoC) will be created. This PoC will implement some form of trivial functionality which should imitate a primitive concept that is required for code-instrumentation tools to work. An example of this would be a simple "hello, world" at the start of a class constructor, or the addition of a simple "hello" and "bye" at the start and end of each method. This will be created by using a model-transformation from the Mod-BEAM plugin as a base, and adding the instrumentations through a MDE transformation. The language to be used for the transformation is likely to be ATL due prior experience or Epsilon / ETL as it seems to be more modern. In the future, a more sophisticated experiment can be performed that will serve to replicate more primitive concepts required for an actual tool to work. The primitive concepts are a result of the research performed in section 5. The result of the PoC can be found in section 5.9.

5 Results

This section contains the result of the research questions as defined in section 3. The answers have been concluded using the research method as defined in section 4. The results section starts off by explaining a set of frequently used frameworks and libraries for code-instrumentation. These frameworks can be considered to be required knowledge on what most tools use in order to function. It is important to keep in mind that there are tools that choose to implement their own form of code-instrumentation. If this is the case, then it will be mentioned in the information for the tool itself. More information on the most commonly used frameworks can be found in section 5.1.

Section 5.2 proceeds by explaining the various usage scenarios for which code-instrumentation is often performed. These scenarios serve to answer the first part of the first subquestion, namely listing the approaches for "Which code-instrumentation approaches and implementations exist for the JVM?". For each of these approaches, a set of concrete implementations have been listed. These can be found in sections 5.3 to 5.7. The list is as-complete-as-possible, containing various frequently used tools. While this list contains most of the important entries that have been found, it remains nigh impossible to complete. This is because there are thousands of entries for the most used libraries that implement code-instrumentation. An example would be how ASM(5.1.1) is used by 1,126 artifacts at the time of writing, while Javassist is used by 1,886 artifacts according to the Maven Repository. Some of the more popular artifacts have been picked out, and special attention has been given to the implementations of tools that handle code-coverage and language extensions, such as aspect-oriented programming, as these approaches are both complex and frequently used. Each of these tools have a detailed description of what they use instrumentation for, and how they achieve this goal. This serves to answer subquestion 2, namely "What technologies and primitive concepts are used in the aforementioned implementations?". The primitive concepts would ideally be a listing of the most basic actions that are required in order for each tool to work as intended with regards to instrumenting or transforming code.

Section 5.8 proceeds by comparing the various tools in each section against another in order to try to determine the answer to subquestion 3, "Is there overlap between these implemented concepts?". Section 5.9 serves as an actual Proof of Concept that Model Driven Engineering can be used in order to implement some of the aforementioned concepts in order to answer the question: "In which manner is model-driven engineering a suitable approach for code-instrumentation?". The answers to these various questions will serve as an answer to the main question of "Which approaches currently exist for code-instrumentation and how do they compare?".

5.1 Frameworks & Libraries

This section contains some of the most-frequently seen code-instrumentation frameworks. Each framework has a simplified explanation on how they work and what they are able to do.

5.1.1 ASM

ASM[2] is "an all purpose Java bytecode manipulation and analysis" framework. Similar to other Java bytecode frameworks, it can be used in order to directly modify existing classes or to dynamically generate classes in binary form. The library can also be used to allow for complex transformations and the building of custom code analysis tools. ASM is used by Cobertura (5.5.2) and Jacoco (5.5.3) to instrument classes in order to measure code coverage.

ASM provides an event-based API (often referred to as the visitor API because of similarities to the visitor pattern) and a tree-based API in order to actually perform the transformations. While ASM is similar to other Java byte-code frameworks, ASM makes it a clear statement that the focus of the project lies in performance.

The visitor API from ASM declares functionality based on the type of event that is visited. These actions in the visitor pattern can be extended, so that additional actions may be performed. Most of the functionality is performed using the ClassReader.accept() function, which makes use of an (extended) ClassVisitor instance. The list below shows an example overview of the various functions that are provided by ASM.

Concept	**Implementation**	**Call**
Find the class declaration	ClassVisitor.visit	ClassReader.accept line 524
Find methods	ClassVisitor.visitMethod	ClassReader.readMethod 1112
Find annotations	MethodVisitor.visitAnnotation	ClassReader.readMethod 1175 (visible), 1193 (invisible)
Find fields	MethodVisitor.visitFieldInsn	ClassReader.readCode 2205
Find labels	MethodVisitor.visitLabel	ClassReader.readCode 2318
Find a specific line	MethodVisitor.visitLineNumber	Only used for labels, traces or manual calls
Find code-jumps (if-else, switch statement, etc)	MethodVisitor.visitJumpInsn and similar instructions	ClassReader.readCode 2040+
Find the end of the method	MethodVisitor.visitEnd	classReader.readMethod 1279
Find the end of the class	ClassVisitor.visitEnd	classReader.accept 683

Table 1: Concept summary of ASM

Each of the mentioned classes have a corresponding writer variant, which contains the functionality in order to actually write the data from the aforementioned visitors.

[2]https://asm.ow2.io/

5.1.2 BCEL

The Byte Code Engineering Library[3] (BCEL) is a library that provides functionality to analyse, create and manipulate (binary) Java .class files. Used by AspectJ (5.6.1) among others.

For the sake of reading in a class, BCEL has a Repository class that allows the developer to look up JavaClass objects. Every retrieved class has an array of iterable methods, which in turn have an array of iterable code. Analysing and instrumenting code is generally done through a visitor pattern. Noticeable of BCEL is that the API of BCEL is extremely low-level, and requires the developer to work with it as such.

During class-loading, the implementation of BCEL itself is triggered by a "JavaWrapper", which in turn creates a "ClassLoader". The ClassLoader has a "ClassLoaderRepository" which is responsible for returning loaded classes as "JavaClass" objects. The JavaClass essentially contains all the data that a Java .class file would have. A JavaClass object is created from a class-file through the usage of a "ClassParser", which reads the following data:

- Header info
- Constant pool, class information and interface information
- Fields, methods and attributes

BCEL allows the developer to set up a set of required tools before instrumentation can be performed. These tools include classes such as the InstructionFactory and InstructionList. The instructions for BCEL need to be performed in such a manner that it feels more similar to writing in assembler rather than working with a Java object. Similar to other bytecode libraries, this functionality requires the knowledge of all possible instructions and flags. Instructions that are supplied to the factory make use of ClassGen, MethodGen and similar counterparts in order to provide the necessary data in an ordered manner.

[3] https://commons.apache.org/proper/commons-bcel/

5.1.3 Javassist

Javassist[4] or Java Programming Assistant, is a Java bytecode manipulation framework. The manipulation of the code is performed at load-time, through a provided class-loader. Javassist has the functionality to perform bytecode manipulation at source-level or on bytecode-level, and is used by Hibernate (5.7.1), SLF4J (5.4.2) amongst others (such as Montric 5.3.2).

Javassist makes use of a "CtClass" (compile-time class) in order to both read existing code and in order to write new instrumentations. Each CtClass is registered to a "ClassPool" container object, so that all instrumentation can be generated at compile-time. CtClass makes use of a "ClassLoader" in order to actually perform the in-depth instrumentation loading of an existing class. By extending this ClassLoader and passing the modified version, extended behaviour can be defined.

Methods can similarly be defined through the usage of the "CtMethod" class. Calls to obtain a method can be performed on the class which should contain a method, throwing an exception if the method could not be found. Furthermore, Javassist has the functionality to create new objects by calling the respective static classes of the desired item. An example would be how a new CtMethod can be defined making use of the static functions of "CtNewMethod". CtNewClass is an exception to this behaviour and does not contain static functions, which allows it to be an actual object so that it can be written as a proper file.

Upon finally writing a .class to disk, the toByteCode function is executed (which is inherited and extended for each CtNewClass). The behaviour of this function differs depending on the existence of a constructor in order to properly handle super classes.

The bytecode functionality of Javassist is slightly more underwater, but necessary in order to function. Functionality that is supported includes objects such as StackMaps in order to verify the produced Bytecode, information on how to read in bytecode, constants and variables and functionality in order to handle annotations amongst others. This functionality is called upon obtaining class from the ClassPool, to analyse the validity of the bytecode and to generate new instructions.

[4] http://www.javassist.org/

5.1.4 Byte Buddy

Byte Buddy[5] is a code generation and manipulation library for the creation and manipulation of Java classes during runtime without the help of a compiler.

Interestingly, Byte Buddy makes use of the visitor API from ASM in order to provide their runtime code generation. Similar to other libraries, Byte Buddy is able to create new classes and redefine existing classes. This is done using a ByteBuddyAgent to perform reloading into an existing JVM, while the ByteBuddy class contains most logic that handles defining and redefining classes. Byte Buddy is used by Hibernate ORM 5.7.1 and is a fairly recent tool with increasing popularity[6]. Byte Buddy aims at making the code "concise and easy to understand". Instead of having to work with a ClassPool or CtClass objects, Byte Buddy works through a straightforward API (from a user-perspective) with the functionality to create a ByteBuddy object, redefine a class or method and then proceeding to write and reload the new object.

[5] http://bytebuddy.net/

[6] https://mvnrepository.com/artifact/net.bytebuddy/byte-buddy

5.2 Identified code-instrumentation approaches for the JVM

This section explains common usages of code-instrumentation and what they are. In order to investigate the implementations, techniques and technologies used with code-instrumentation, we first have to identify for what purposes code-instrumentation is performed. The identified approaches are as follows:

- Profiling: Profiling is a form of code analysis that is used to measure aspects of the software. Through the use of profiling, aspects such as time between actions, the frequency of actions or the complexity of code can be measured. Profiling requires the execution of code in order to work, as additional profiling code is injected into the code (either the source-code or bytecode). Object-size estimation & Memory Analysis can also be considered to be possible using instrumentation, but both are often included in profiling tools in the modern day. Object size calculation can be performed to calculate the size of objects that are currently active in the JVM. This can be used to, for example, debug memory leaks or find memory bottlenecks in the application. Memory analysis is performed in order to check the current memory usage of an application. Standalone tools often create a snapshot of the current memory status without using instrumentation, such as the Eclipse Memory Analyzer[7].

- Logging & Tracing: Through program logs, more information can be gathered about the executing software. Software tracing is slightly different than profiling, as it allows for paths through the code to be logged. Software traces are very similar to logging, and both forms are often used as a form of program debugging.

- Code-coverage analysis: Slightly more complex is the analysis of code-coverage. This is done by looking at testing frameworks and mapping them to the actual functions in the source code. The coverage requires information on what functions there are, what functions are being tested and especially which code inside each function is actually reached during the tests.

- Dynamic code-analysis: Code-analysis is as the name implies the analysis of an entire program. More complex actions for code-analysis might include null-pointer dereferencing. Tools that are able to analyse entire programs without producing false-negatives or false-positives are popular and often used as a part of a development stack. Different from static code analysis where the analysis is performed without executing the code, dynamic code analysis requires the execution of the program in order to function. Dedicated code-analysis tools will not be covered in this research, as static tools have been researched before. I was unable to find dynamic code-analysis tools for Java applications that were suitable for research.

- Aspect-oriented programming: Aspect-oriented programming and other language extensions stand out as they are different from other common uses of code-instrumentation. Aspect-oriented programming uses the concept "aspects" in order to generate modular code without changing the original code. This code is then "weaved" in the bytecode in order to create a runnable application.

- Language extensions: Besides conventional Java usage, frameworks and the likes have been created in order to allow for extensions on the Java language.

[7] https://www.eclipse.org/mat/

5.3 Profiling

This section lists tools that are able to perform profiling using code-instrumentation.

5.3.1 VisualVM

VisualVM[8] is a popular profiling tool for Java applications that includes the functionality to create an accurate overview of CPU usage per function and memory usage amongst others. The tool makes use of instrumentation for various parts of the functionality that the tool provides. The actual instrumentation is implemented in a library called **JFluid**[2], which was developed as part of the NetBeans profiler by Sun Labs[9].

An example of how VisualVM makes use of instrumentation would be their CPU statistics display. The visual side of the display is handled in the "CPULivePanel" class in the profiler package. This panel makes use of a "LiveCPUViewUpdater" in order to refresh the results. This updater, in turn, has a JFluid "ProfilerClient", which provides a set of Java profiling functions. Upon update, the method "getCPUProfilingResultsSnapshot()" is called, which literally provides a snapshot of the current profiling results. This function simply reads back data from the instrumentation that was registered before this point.

At start-up, JFluid creates a CPUCallGraphBuilder in order to mark all important points where profiling should be inserted. The "ClassManager" is able to filter these points based on which class they reside in. These classes are then put into a batch that need to be instrumented.

The actual instrumentation is called from the "ProfilerClient". Some basic actions that are required at the start are to load the root class and to load follow-up methods. A "RecursiveMethodInstrumentor" is then used to collect which data should be instrumented. An "InstrumentMethodGroupData" class is then called to write the actual instrumentation.

Some key concepts from this that can be found are:

- Parse the code to an internal Java format (performed by JFluid)
- Find the root (main) classes (provided by VisualVM)
- Find the root (main) method(s) (implemented in RootMethods.java)
- Find all methods (Implemented by inheritors of AbstractDataFrameProcessor.java)
- Find key-points to be instrumented (CPUCallGraphBuilder.java)
- Instrument the code (performed by JFluid)

It should also be mentioned that the way instrumentation was implemented was very custom-tailored for the scenarios of the multi-threaded JFluid profiler, which very much reduces the readability of how the various actions are performed. While the classes of RootMethod and AbstractDataFrameProcessor are still of manageable size (around 100-200 lines of code), the CPUCallGraphBuilder alone is about 1500 lines of code. The RecursiveMethodInstrumentor that is used to collect the necessary to-be-instrumented data is about 700 lines of code. Combining this with the client-server architecture and abundance of synchronized methods, this tool would be very time-consuming to analyse in more detail.

[8] https://visualvm.github.io/
[9] https://profiler.netbeans.org/jfluid.html

5.3.2 Montric

Montric[10], previously also known as EurekaJ, is an open-source Java profiler that is quite far in development. Similar to other profilers, it contains functionality to monitor CPU- and heap usage, amongst others. Montric is a profiler that makes use of a combination of an Ember client, REST as intermediary and a server written in Java. Due to lack of news in recent years, the project can be considered to be on hiatus, albeit it still has most functionality that one would expect of a profiler. The Montric server makes use of a combination of native Java instrumentation functionality and **Javassist** in order to obtain the desired results

Instrumentation is performed in the "org.eurekaJ.agent" package, starting with the "EurekaJTransformer". The "transform" method is the initial called function that starts off the process of redefining a class. Data is read in as a bytestream and transformed into a Javassist "CtClass" object. Additional data is read out during the transformation process in order to generate "ClassInstrumentationInfo" which contains track of the package name, class name and if the class extends or implements other classes. Montric tries to obtain the uppermost level class to be instrumented in order to check if the implemented interface or abstract classes can be instrumented. Montric then proceeds by going through all declared methods (CtMethod) from the CtClass object, and runs instrumentMethod on each of these methods.

The way that a method is instrumented depends on the type of method. As of the time of writing this report constructors, getters and setters are not yet supported. It can be assumed that these methods would provide an overflow of data or are not properly parsed yet on the client-side of the application. A usual case of instrumentation continues to the "addTiming()" function. This method quite plainly appends timing information to the method that can be used to measure the execution performance.

What stands out from the rest is that the transformer makes use of a loadClassInClassLoader function that adds classes to the loader if this was not done yet. This seems to be an important prerequisite in order to support JEE environments, as is noted in the comments.

Some key concepts from this that can be found are:

- Parse the code to an internal Java format (Javassist)
- Find all methods (visitor pattern)
- Find the highest-level abstraction (ClassInstrumentationInfo)
- Find the lowest-level abstraction (ClassInstrumentationInfo)
- Identify the method type (Javassist)
- Instrument each method on the correct abstraction level (EurekaJTransformer)

[10]https://github.com/joachimhs/Montric

5.4 Logging & Tracing

This section lists tools that are able to perform logging and tracing using code-instrumentation.

5.4.1 BTrace

BTrace[11] calls itself a "safe, dynamic code tracing tool for Java". BTrace works through the use of dynamic bytecode instrumentation, inserting traces into an active Java application and then hot-swapping the traced program classes. In order to perform the actual instrumentation, BTrace makes use of the **ASM** bytecode library.

Btrace allows users of the tool to add annotations that define where to add a "probe" into the program. An example would be an interest in the class "java.lang.Thread" and the method "start()", which would add probes for each entry into "Thread.start()". A function can then be defined which specifies what behavior to execute when a Thread.start() has been called. It is notable that BTrace contains a "hacked" version of ASM in their own source-code. This "hacked" version allows for some custom modifications that the creators of BTrace found to be necessary, such as merging instrumentation invocations.

In essence, BTrace works by gathering all the probes that have been defined. These probes are collected in the "runtime" package of the tool. This package also specifies what data should be stored for a probe (in the "OnProbe.java" file) and what the bytecode of the probe should look like (in the "BTraceProbeNode.java" file). The actual injection of the bytecode contains the most expansive class, as the "Instrumentor.java" file contains 1743 lines of code, with most of it defining what to do when a specific OP-code has been encountered. A reason for the expansiveness of this instrumentation could be the support for complete CallGraphs and call traces that come from a specific probe.

The functionality for BTrace can be separated into two parts. The first is weaving instrumentations into executing code based on the user-defined hook points. The second is to create a callgraph of the hook-points in order to create the logging statistics.

The functionality requirements in order to identify and weave probes can be summarised as follows:

- Find all probe points (BTraceProbe)
- Gather the instructions to execute on probe points (BTraceTransformer)
- Instrument the existing code

The functionality required in order to build call-graphs can be summarised as follows:

- Compute a complete trace-log from each probe point (CallGraph)
- Gather instructions to execute on each action in order to create a trace report
- Instrument the existing code

Similar to tools mentioned before this, BTrace also requires the functionality to read and write the actual bytecode and perform the necessary hot-swapping. All of this is done by extending the functionality of ASM.

[11] https://github.com/btraceio/btrace

5.4.2 SLF4J

SLF4J[12] or "Simple Logging Facade for Java"[13] is a façade or abstraction of various logging frameworks, allowing developers to plug a logging framework to their application at the time of deployment. The logger for SLF4j is customizable and allows for easier output to files or custom event handlers. SLF4J makes use of **Javassist** as an optional compile dependency in order to extend logging to the entire application as opposed to select events[14].

The instrumentation in SLF4J is, as expected, extremely simple. This is because the goal is to instrument each method with simple logging information. The actual instrumentation is performed in the "LogTransformer" of the "org.slf4j.instrumentation" package. The logging is added on load-time. In SLF4J, the code transformation is processed in the "transform()" function. This function quite simply calls the "doClass()" function for each available class. A class is then adjusted to contain the appropriate logging information (such as a "logger" instance). For each method (currently only non-empty, defined as "CtBehavior" from the library), the "doMethod()" function is called which adds an action using "method.insertBefore()" and "method.insertAfter()". The data to be added is obtained by retrieving the signature and the return value of the method. No timing information is added, as slf4j is a logger and not a profiler.

Important concepts can thus be summarized as:

- Find class and method information (such as the name)
- Find the start of each method (visitor pattern)
- Find the end of each method (visitor pattern)
- Instrument every method

[12] https://github.com/qos-ch/slf4j
[13] https://www.slf4j.org/
[14] https://mvnrepository.com/artifact/org.slf4j/slf4j-ext/1.7.25

5.5 Code-coverage analysis

This section lists tools that are able to perform code-coverage analysis using code-instrumentation.

5.5.1 Clover

Clover[15] is a tool for Java and Groovy that aims to provide reliable code-coverage analysis. Clover makes use of source-code instrumentation, meaning that they directly add instrumentations to the source-code. Clover makes use of their **own implementation** in order to perform these instrumentations.

Clover stores all data retrieved from the project in a database. This database is known as "Clover.db"[16] before usage, and is used internally as the "Registry". The registry file is written during the instrumentation process, and overwritten when a new instrumentation process is executed. The registry is read by the Clover-instrumented code when it is executed, during report generation or during coverage-browsing (such as through the IDE)[17]. This database contains the information about the entire project, including classes and methods. Clover makes use of a set of predefined types listed as a set of "reservedContexts" that identify the various types of reserved names, such as for/while loops or if/else statements among others.

The preparation code for the code-instrumentation that Clover uses starts by gathering miscellaneous data in order to start the instrumentation process. Such data includes the gathering of the amount of lines and the encoding used. In order to parse files, Clover makes use of JavaLexer and their own filter in order to hide whitespace and comments. This is used in order to parse the code to native Java classes that can be used for instrumentation. After generating the filter, the actual instrumentation is performed. The code-instrumentation makes use of the "CloverToken" class which contains a previous and next token, but also a set of pre- and post-emitters. These emitters are usually existing instructions, lambda expressions, method entry- or exit-points or class entry- or exit-points[18].

The primitive concepts of identifying classes and methods are implemented by Clover.db, using their own filter and by parsing the Java code. The instrumentation concepts of identifying statements and test-methods are then performed using the implemented CloverToken and emitter system. In order to work, Clover has to define the type of each emitter in a token. This can be either a pre- or post-emitter. Each of the emitter types are defined as a class that inherits from "Emitter" itself. Required functionality can be summarised as:

- Parse Java (JavaLexer)
- Create a database that records instrumented functions (Clover.db)
- Find the start of a test method (add pre-emitters)
- Find the end of a test method (add post-emitters)
- Instrument called functions
- Instrument the test code

[15] https://www.atlassian.com/software/clover
[16] https://confluence.atlassian.com/clover/database-structure-420972456.html
[17] https://confluence.atlassian.com/clover/managing-the-coverage-database-72253456.html
[18] http://trojmiasto.jug.pl/wp-content/uploads/2013/05/Clover-presentation-for-JUG-Trojmiasto.pdf

5.5.2 Cobertura

Cobertura[19] is a Java tool that can be used to calculate the amount of code that is covered by tests. It makes use of offline byte-code instrumentation. Cobertura works by loading in the byte-code of the software in the memory, performing instrumentation on the classes that were supplied. Cobertura makes use of **ASM** in order to perform the actual instrumentation for the code.

The pipeline of actions of Cobertura can be found in the "Cobertura.java" file. The constructor starts off by creating an empty report, reading in the arguments and creating an instrumentation task, a coverage task and a project data merge task. Of those tasks, code instrumentation is quite obviously handled starting with the CodeInstrumentationTask This task creates a "CoberturaInstrumenter", preparing it in order to run for each class that was provided in the arguments. Before the actual instrumentation begins, a "ClassPattern" instance is generated in order to setup the required regex settings to check for matching class names. Along with this, more settings are provided for the CoberturaInstrumenter itself by reading in all arguments. The instrumentation itself is called in the various addInstrumentation functions from the CodeInstrumentationTask. There exist various functions that provide different results, depending on whether the output needs to be stored in an archive or not. Instrumentation functions in the CoberturaInstrumenter each make use of an InputStream of the original file. The file is always converted before the actual instrumentation begins.

The comments in the CoberturaInstrumenter mention that a file is instrumented in three different passes, as noted below. In each of these three steps, it is noticeable that each of these classes inherit the functionality of the "ClassVisitor" from ASM. This ClassVisitor contains all the functionality to visit practically each possible part of a class, returning the properties that were requested per method. An example would be that the visitMethod function would return the access flags, Opcodes, name, descriptor, signature and exceptions of a method. The base visit function of each class is triggered in the "accept" method of the "ClassReader" class from ASM.

The first pass performed by Cobertura is to make use of the DetectDuplicatedCodeClassVisitor, which is an extension of the ClassVisitor from ASM. It is used in order to detect duplicate code and store the data of where it was found. The class overrides the original visitMethod function from ASM, so that the function first obtains the original method data from the original ClassVisitor, before proceeding to create an instance of the "DetectDuplicatedCodeMethodVisitor" class. This class keeps track of the actual duplicated lines found. It makes use of a "CodeFootStamp" in order to keep track if two lines are identical or not, giving the visit of a footstamp priority, before proceeding to visit a line, label or other item depending on the function called. As an end result, the DetectDuplicatedCodeClassVisitor contains a mapping of line numbers (keys) with the duplicated lineId and origin lineId as values.

The second pass is the BuildClassMapClassVisitor, which finds all touch-points and other interesting information that are in the classes, and subsequently stores it in a ClassMap. Data that is stored contains the original class name and the original source code of the class. At the stage that the visitMethod function is called, it is checked if the class is yet to be instrumented. If this is the case, then a FindTouchPointsMethodAdapter is created. This adapter analyses a given method, which practically traverses the instructions in a method line by line, jumping when necessary. The goal of this step is to gather "Touch-points", which are points in the source code that Cobertura wants to monitor for usage in the third and final step. Touch points always contain an eventId and a lineNumber, and can contain more data depending on if it is a "JumpTouchPoint", "SwitchTouchPoint" or "LineTouchPoint".

The final step of the various passes performs the "real" instrumentation. It is implemented in the InjectCodeClassInstrumenter, which makes use of the previously generated ClassMap in order instrument the code.

[19] http://cobertura.github.io/cobertura/

The items below are a summary of the primitive concepts that are used by Cobertura in order to perform code-coverage analysis:

- Parse Java (ASM)
- Find dulicates (DetectDulicatedCodeMethodVisitor)
- Find touch-points (BuildClassMapVisitor and FindTouchPointsMethodAdapter)
- Insert calls (InjectCodeClassInstrumenter)

5.5.3 JaCoCo

JaCoCo[20] is a code-coverage analysis library for Java. It makes use of on-the-fly bytecode instrumentation. Similar to Cobertura, JaCoCO makes use of **ASM** in order to implement instrumentation. The initial step is also similar, as the input classes are first transformed into an InputStream. This is performed in the "Instrumenter" class in the "core" package.

This instrumenter class creates the necessary readers and writers and proceeds by creating a "ProbeArayStrategyFactory". This class will essentially check if the given class is either a Module/Interface or a normal class. In the case of a normal class, it will create a "ClassFieldProbeArrayStrategy".

The generated strategy is used in order to "create a static field to hold the probe array and a static initialization method requesting the probe array from the runtime", this means that it is used in order to properly access and instrument static fields and methods. The instrumenter then proceeds by creating a "ClassVisitor" from ASM in the form of a "ClassProbesAdapter" as written by JaCoCo. This visitor accepts a delegated "ClassInstrumenter" with the previously created strategy.

The "ClassInstrumenter" implements the actions that should be performed upon visit, field visit and method visit and returns an instrumented version of the visited classes. The instrumented classes contain probes as created using a "ProbeInserter" on method visits. These probes are used to add probes into the control flow of a method, so that the code can be marked as executed or not[21]. In the visitMethod function of a ClassInstrumenter, a new MethodInstrumenter is created in order to actually visit the various instrumentations of that method. This method is then similarly filled with probes where needed. The ClassProbesAdapter that delegates the visitor is used to calculate probes for each method, and to keep track of the current probeId. These probes can then be used for the code-coverage analysis itself.

It is noticeable that JaCoCo implements the instrumentation using probes by itself, and makes use of the visitor pattern by ASM in order to traverse through the actual code. The table below contains all functions that are called from the "classVisitor". JaCoCo defines two method visitors that are called in the overridden visitEnd function from ASM. The first is to eliminate consecutive duplicate frames, making sure that the generated code is actually valid. The second is the probeInserter which actually performs the tracking and inserting of probes. For the "DuplicateFrameEliminator" The functions are all identical to the original, only setting a boolean for "instruction" to true or false and otherwise acting as a passthrough.

Below is a summary of the concepts that JaCoCO requires in order to perform code-coverage analysis:

- Parse Java (ASM)
- Gather probe points (ClassProbesAdapter)
- Eliminate "consecutive duplicate frames" (DuplicateFrameEliminator)
- Track and insert probes (ProbeInserter / ClassInstrumenter)

[20] https://www.eclemma.org/jacoco/
[21] https://www.jacoco.org/jacoco/trunk/doc/flow.html

5.6 Aspect-oriented programming

This section lists tools that are able to perform aspect-oriented programming using code-instrumentation.

5.6.1 AspectJ

AspectJ[22] is an aspect-oriented extension to Java that supports general-purpose aspect-oriented programming. The goal of AspectJ is to allow for clean modularization of cross-cutting concerns, allowing for both design modularity and source code modularity. AspectJ makes use of their own, extended version of **BCEL**.

AspectJ allows for modularization through the usage of a Java overlay, namely dynamic join points. These dynamic join points are described as "points in the execution" of Java programs. The additions are pointcuts (selective join points and values at those points), advice (additional actions to take at join points in a pointcut), inter-type declarations ("open classes") and aspects ("a modular unit of crosscutting behavior, comprised of advice, inter-type, pointcut field, constructor and method declarations"[3]). AspectJ implements these various extensions of the Java programming language by allowing the developer to program the functionality, and then "weaving" the newly added code to the original program.

The AspectJ compiler accepts both sourcecode as well as bytecode, and returns the newly formed bytecode as a result [4]. The compiler has been mentioned to be a 2-stage process. The first stage compiles both AspectJ and pure Java source code into pure Java bytecode annotated with the additions mentioned earlier. The back-end consists of the actual transformations encoded in these attributes in order to produce the final .class files.

As mentioned earlier, AspectJ makes use of their own extended version of BCEL. This is done as the original BCEL is not actively developed anymore and does not intergrate the patches that were made by the AspectJ development team[23]. As mentioned earlier, the weaving process in AspectJ makes use of dynamic Join Points. These Join Points contain respective "shadows" in the original sourcecode or bytecode of the application. These static shadows may be of different types, where each shadow represents a location where the new code should be inserted. In the weaving process, each identified shadow is of the type "BcelShadow", which can be found in the "weaving" package. Each shadow has a "Kind", which may be a method, constructor or nearly any other Java type.

The weaving process usually starts in the "WeavingURLClassLoader" in the "weaver.loadtime" package. The classloader proceeds by making a "WeavingAdaptor" from the "weaver.tools" package. This class then creates a "BcelWorld" object and a "BcelWeaver" with the "BcelWorld" as reference. This "BcelWeaver" contains the functionality to find pointCuts, optimize them, detect duplicity and to actually weave them together.

AspectJ also looks for an ASM library in its weaving process in order to add stack map attributes to the produced bytecode. This is done using ASM as BCEL did support this functionality, but is severely dated. Stack maps are used in order to allow for verification of the produced bytecode.

The various concepts of AspectJ can be summarised in the following list:

- Parse into Java code (BCEL adaption)
- Gather join points (BcelShadow)
- Weave the new code into existing code (BcelWeaver)

[22] https://www.eclipse.org/aspectj/

[23] https://www.eclipse.org/aspectj/doc/released/faq.php#q:bcel

5.6.2 Compose*

Compose*[24] (Compose-star) is a Java project that mainly focuses on aspect-oriented programming. The goal of the project is similar to AspectJ, with the aim to "enhance the modularization capabilities of component- and object-based programming". The initial goals for the project upon research were to "provide a framework to experiment with new language concepts & features" and to "provide the ability for researhers and practitioners to apply the Composition Filters language"[1]. While Compose* is cross-platform, this research will only take into account the parts that are used for the Java instrumentation process. Compose* has adopted **Javassist** in order to provide concise Java instrumentation.

Compose* defines two important concepts in order to perform instrumentation. These concepts are the filtermodule and the superimposition. The filtermodule essentially defines what class or methods should be created and which functionality they provide, while the superimposition defines how this new class should be linked. Superimpositions and filters are essential in order to determine how a HookPoint in a class is defined, as can be found in the "getMethodInterceptions()" function of the "JavaWeaver" class. The actual weaving process is started off in the the "JavaWeaver" class, situated in the "ComposeStar.Java.WEAVER" package. Compose* makes use of instrumentation in order to weave selected hooks at compile-time[5].

As mentioned before, the instrumentation process for weaving is started in the JavaWeaver. The function "run()" is responsible for kicking off this process. At first, a "HookDictionary" is created. These hooks are all the locations where a call to the interpreter should be inserted by the weaver. The HookDictionary contains hooks for method calls and instantiations. The data for this is loaded in using the CommonResources object that is passed through to the function. A "ClassWeaver" is instantiated, and each file contained in the "CommonResources" has their classpath added to this weaver. This process is then continued in the "weave()" function from the "ClassWeaver" class.

The actual weaving process adds the processed class to the resoures file and creates a new directory. It makes use of the CtClass objects from Javassist (5.1.3). The process itself adds all classes-to-weave to a HashSet, after which an initializer class is generated which allows the class to initialize runtime of the class without optimizing it away by a Just-In-Time (JIT) Compiler. Each weaved object is then output to the corresponding file(s).

The weaving process can be summarized as follows:

- Parse into Java code (Javassist)
- Gather hook points (Create a dictionary of hook-points) (Compose*)
- Find all classes and methods (Javassist: ClassPool, CtClass)
- Instrument methods by appending created code (constructors, method initializers) to existing class files (Javassist: CtConstructor)

As Compose* is split up between a multi-language CORE and a Java-specific part, it should be noted that most of the actual code-generation is done at the CORE-level, while the weaving is done at the Java-specific level.

[24] http://composestar.sourceforge.net/

5.6.3 JAsCo

JAsCo[25] defines itself as an "advanced aspect-oriented programming language", which was originally tailored for the component-based field. The official website also mentions that it supports a wide range of established AOP functionalities, such as AspectJ-like pointcuts. JAsCo introduces the concept of "aspect beans" and "connectors" as the main new concepts to work with. A major difference between JAsCo and the other AOP languages mentioned in this research is that JAsCo does not have the code available for study. It is mentioned that the code has been integrated in a tool called PacoSuite, but this tool is at the time of writing severely dated. It is mentioned in one of the original research papers that JAsCo makes use of a **combination** of their **own** bytecode-manipulation library, in combination with **BCEL** to "transform Java byte-code to human readable Jasmin assembler code"[6] (and the other way around).

As mentioned earlier, the idea behind JAsCo is the usage of "aspect beans" and "connectors". An aspect bean defines execution behaviour through the usage of "hooks". The original goal of JAsCo was to make use of these new concepts in order to not only integrate AOP with object-oriented programming, but to also integrate AOP with component-based software development, as it tends to suffer from similar issues. At the time of writing for the official report on JAsCo (2003[6]), JAsCo aimed to resolve the problem of non-reusability of aspects as used in AspectJ, while staying true to the native Java syntax. A hook in JAsCo has a simple syntax, containing a "when" and "what" component, for example to execute on a certain method, and to define extra functionality on the call of that method. A connector can then in turn instantiate an actual hook, or multiple hooks for that matter. Doing so creates some form of control-flow which allows for readable and reusable code.

The various connectors (containing hook instances) would be registered to a Connector registry, and all that would remain at this point would be to actually weave the code together. This could be performed at runtime, allowing for flexible code-changes. It can be assumed that the weaving process would need the following instrumentation functionality from a library in order to support the desired functionality:

- Find all classes and methods
- Gather hooks and connectors
- Allow users to create their own instrumentations in a readable format
- Append created code to code at hook-points

The actual management of hooks and connectors would be handled by JAsCo itself, making use of mentioned tools such as "CompileConnector" (add to classpath), "RemoveConnector" (remove from classpath), "CompileAspect" (compile an aspect to a normal Java bean) and "Introspect" (allows for GUI introspecting which connectors are loaded for a given classpath, and which hooks are instantiated).

[25] http://ssel.vub.ac.be/jasco/what.html

5.7 Other usages

This section lists tools that make use of code-instrumentation for more miscellaneous tasks.

5.7.1 Hibernate ORM

Hibernate ORM is a object/relational mapping framework for Java applications. While Hibernate does not use instrumentation for its core functionality, it does require instrumentation for its usage of proxies. This is used in order to implement lazy-loading, so that classes are only loaded in memory when they are first used, as opposed to being loaded immediately. The proxy design pattern can alternatively also be used to implement eager loading. Hibernate currently makes use of **Byte Buddy** in order to implement proxies[26]. Hibernate initially made use of Javassist in order to implement instrumentation, but has migrated to Byte Buddy in November 2016. The source-code for Hibernate ORM currently contains both the code for Javassist and the code to run with Byte Buddy in the various org.hibernate.bytecode sub-packages. The actual proxy code is situated in the "org.hibernate.proxy" package.

The actions required for the lazy initialization of "POJOs" or "Plain Old Java Objects" are stored in the abstract "BasicLazyInitializer" class. This class is inherited by the "ByteBuddyLazyInterceptor" and the "JavassistLazyInitializer" and contains placeholders for data storage, invocation and replacement (if a class is loaded but not set in the proxy). The most noticable difference between the ByteBuddy and Javassist implementations are that the Javassist implementation seems to focus on the "invoke" function of a class, while the ByteBuddy (current) implementation overrides the "intercept" functionality. As there are currently no more calls to Invoke, while Intercept is actually being used, it would seem that ByteBuddy would allow for a better guarantee that all class calling is handled through proxies, instead of having to check a class has been set in the proxy or not. Invoke from Javassist works through the use of the MethodHandler super class, while the intercept functionality from ByteBuddy works through the usage of the @RuntimeType annotation.

The concepts that Hibernate uses for proxies can be summarised as follows:

- Intercept method calls (lazy initialize in the old Javassist version)
- Keep track of loaded classes (only required in the old Javassist version)
- Invoke the proxy
- Deserialize the proxy

Hibernate also makes use of ByteBuddy for the purpose of "Enhancement". Enhancement includes "lazy state initialization, dirtiness tracking, automatic bi-directional association management and performance optimizations"[27]. This enhancement is, however, disabled per-default, and requires functionality that is similar to that used in aspect-oriented programming languages in order to weave instrumentation in the original code.

[26] http://in.relation.to/2016/11/18/meet-rafael-winterhalter-and-bytebuddy/
[27] https://docs.jboss.org/hibernate/orm/5.0/topical/html/bytecode/BytecodeEnhancement.html

5.8 Identified overlap

This section describes the overlap between identified concepts within the several investigated approaches of section 5.2. These concepts will first be compared to tools within their own categories, and are compared with other categories where appropriate.

- Profiling

The comparison in profiling tools quite simply consists of the comparison between VisualVM (5.3.1) and Montric (5.3.2). As a short recap; VisualVM is a fully fledged profiling tool that internally makes use of JFluid in order to handle most of its functionality. JFluid is also responsible for parts that contain code-instrumentation, while making use of their own implementation to perform the actual instrumentation. Montric, on the other hand, primarily uses Javassist in order to maintain clearly structured and divided code.

While the implementations of both tools may be extremely different from eachother, there are concepts that are shared between the two tools. It can be reasoned that this is because the end-goal of the tool is quite obviously the same, namely to visualize profiling results for a running Java application. Unfortunately, profiling tools are very broad so it would be an in-depth research on its own in order to identify how all parts of the profiling functionality are handled. With the focus on CPU call timings, it can be summarised that some basic instrumentation functionality is required; classes and methods need to be parsed into an internal format in order to find the key-points that should be instrumented. Furthermore, it is required that additional instructions are added to the code in order to represent timing data. The new code will have to be hot-swapped, or it would be impossible to toggle profiling on- or off.

- Logging & Tracing

The comparison between tools of this section is between BTrace (5.4.1) and SLF4J (5.4.2). BTrace is a tool that supports tracing and the ability to inject code into specific function calls. BTrace makes use of ASM in order to perform dynamic code instrumentation. SLF4J on the other hand makes use of code-instrumentation for simpler means. SLF4J makes use of Javassist in order to directly add logging into quite literally every method. As such, SLF4J has no need to gather hooks, points or anything complex that determines where code should be added. Instead, it can simply use the visitor pattern provided by the instrumentation library in order to traverse every method and add the necessary logging information. BTrace on the other hand performs vastly more complex actions in order to detect where code should be added and what code it is that should be added at that those specific points.

It is quite notable that BTrace shares the requirement of having to detect certain hooks or probes with various other tools, while also sharing the necessity of having to be able to add user-defined instrumentation. SLF4J only requires basic print statements in order to function.

- Code-coverage analysis

One of the easiest comparisons to make for code-coverage analysis is the comparison between Cobertura (5.5.2) and JaCoCo (5.5.3), as both tools make use of ASM in order to implement the instrumentation functionality. It is noticeable that both tools make use of ASM in order to, first of all, remove duplicate code. While the implementation slightly differs on how the visitor pattern is extended, both tools perform this functionality before actually instrumenting touchPoints or probes (depending on the tool). As both tools use some form of touchPoints or hooks in the code to analyse the executed actions, the concepts used are also very similar. Both tools require knowledge about the basic Class information, method information and field information. Both Cobertura and JaCoCo require extensive knowledge about the actual content, and override the various action-based visitors in order to insert the probes and touch points on the desired locations. Both tools share a common interest in eliminating duplicate code. Without prior knowledge, a first guess would be to use this data in order to analyse duplicate code for the user. In the code of JaCoCo it is however mentioned that the removal of duplicate code is for the sake of generating valid instrumentations. This has to do with the duplicate generation of try/catch statements, which could likely be something that is generated when using the visitor pattern fro the framework.

Clover (5.5.1) is slightly different from the other two, making use of its own implementation for source-code instrumentation. Clover makes use of Tokens and emitters instead of touchPoints or probes. The code still requires knowledge of the various forms of instructions, the entry and exit

of methods and of course the entry and exit of a class itself. There is no clear mention to be found on having to eliminate duplicate code.

From this it can be concluded that the various coverage-analysis tools all require extremely similar concepts for the sake of monitoring important points in the execution of the final program. All these tools require knowledge about the classes, methods and instructions within these methods.

- Aspect-oriented programming

The first comparison to make would be between AspectJ (5.6.1) and Compose* (5.6.2). AspectJ makes use of an extended implementation of the BCEL framework, while Compose* makes use of the more modern Javassist framework. Regardless, both tools make use of a system that requires "join points" in the case of AspectJ and "hook-points" in the case of Compose*. These points are tracked by the tools themselves in order to later determine how the code should be weaved together. The weaving process is also adapted separately by each tool, which would indicate that there is no support in the libraries that allow for easy integration of large amounts of code with the existing code-base. This also counts or JAsCo which also makes use of "hooks" in order to link "aspect beans" and "connectors".

All three tools require the full set of instrumentation functionality, as users of the tool will want to be able to add any kind of instruction similar to how one would write normal code, but in an aspect-oriented manner.

- Other usages

While there may only be one tool in the Other Usages section, namely Hibernate ORM (5.7.1, we can still compare it as it contains references to two libraries. The first library is Javassist, which was later phased out in favor of Byte Buddy. As the usage for Hibernate mainly focuses on proxies, we can compare what concepts were needed in order to perform this functionality and if those concepts are shared or not. As mentioned in the analysis of the tool itself, it was notable that instead of invoking a proxy, the tool now intercepts, at which point it creates a proxy instead. The required concepts were also identical, being able to be summarised as requiring the creation of a proxy, keeping track of the proxy (mainly used for Javassist), and being able to turn the proxy back into a normal, usable class.

Different from Hibernate, the frameworks themselves also contain quite the obvious forms of overlap. All frameworks listed in section 5.1 contain the ability to somehow navigate through various classes and methods in a structural order. The frameworks allow the developers to adjust what happens at any point of a class, method or instruction and each framework allows for the creation of additional instructions that range from calling an imported package to creating a completely new class. The more older frameworks such as BCEL 5.1.2 have a tendency to let develop their API to be extremely similar to bytecode itself, while more modern frameworks such as Javassist (5.1.3) and Byte Buddy (5.1.4) or even ASM (5.1.1) attempt to make the API as close to "normal" code as possible.

5.9 Suitability of MDE for code-instrumentation

As mentioned in section 4.3, the Proof of Concept consists of a simple MDE transformation that makes use of the Mod-BEAM tools and metamodel. These resources are both provided for this research. The setup process is as follows: A simple Java file with a main method and a constructor is provided as input file. This file is processed by the Mod-BEAM tools to automatically create a .jbc model that adheres to the rules of the Mod-BEAM metamodel. A .class file is generated from the .jbc model in order to run the file as a Java application.

What the PoC introduces is a transformation from the tool-generated .jbc model to a new .jbc model that contains additional instructions. The transformed code adheres to the same metamodel standard, and is transformed using a MDE transformation language. The additional instrumentation consists of a plain System.out.println() with the text "IN: " followed by the name of the calling method. This instrumentation is added at the start of every method, called or not. Listing 1 shows what the original input should be that is used to generate the .jbc object. Listing 2 shows what we want to work towards. Image 1 shows what our output should look like.

```
1  public class InputClass {
2      public static void main(String[] args) {
3          new InputClass();
4      }
5
6      public InputClass() {
7          usedMethod();
8      }
9
10     private void unusedMethod() {
11
12     }
13
14     private void usedMethod() {
15
16     }
17 }
```

Listing 1: InputClass.java

```
1  public class DesiredClass {
2      public static void main(String[] args) {
3          System.out.println("IN: main");
4          new DesiredClass();
5      }
6
7      public DesiredClass() {
8          System.out.println("IN: DesiredClass constructor");
9          usedMethod();
10     }
11
12     private void unusedMethod() {
13         System.out.println("IN: unusedMethod");
14     }
15
16     private void usedMethod() {
17         System.out.println("IN: usedMethod");
18     }
19 }
```

Listing 2: DesiredClass.java

```
<terminated> InputClass (1) [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe
IN: main
IN: InputClass constructor
IN: usedMethod
```

Figure 1: Desired output

In order to do this, the following steps would have to be executed in a model-transformation language. These steps would remain the same independent of the transformation language used. In order to add a print statement at each method entry:

- Store the initial instruction
- Store the follow-up instruction
- Add the new code, consisting of a GetStatic, LDC and Invocation instruction
- Link the three new instructions together using unconditional edges
- Link the initial instruction to the first new instruction
- Link the last new instruction to the original followup instruction

The first iteration of the PoC made use of ATL (ATLas) as MDE transformation language. The advantages ATL are that it is declarative, so that each transformation has a clearly defined ruleset. Another advantage would be access to "refining" mode, which automatically copies all original model data to the target model for items that have no specified rules, as long as the source- and target metamodels are the same. This turned out to be a dead-end fairly soon after, as the syntax had some unclear parts when trying to define new child-objects that are an instance of a metamodel type rather than a primitive type.

The second, current iteration of the PoC makes use of QVT (operational) which is a more imperative model-transformation language. Rules are defined through mappings, which can be called in a slightly more hierarchical manner rather than purely declarative rules. QVTo has a syntax which is more similar to a common programming language, which made it far more simple to define the required instructions. Copying of the original data was possible by simply executing a deepclone() at the initial stage, after which any additional refinement mappings could be executed on the destination model.

A comparison of the .jbc input and .jbc output and desired .jbc code is visible in image 2. It can be seen that the appropriate instructions have been generated. The order in the file slightly differs, but this is no problem as the Unconditional Edge properties are responsible for chaining instructions together. From the generated .jbc, a .class file can be generated which leads to the exact same results as depicted in image 1. A simplified version of the transformation code can be found in appendix A, where the helpers that create the instructions have been removed in order to improve readability. The code in the appendix depicts the basic order of operations that is required in order to generate the correct output. The full code for this project is attached with the original version of this document.

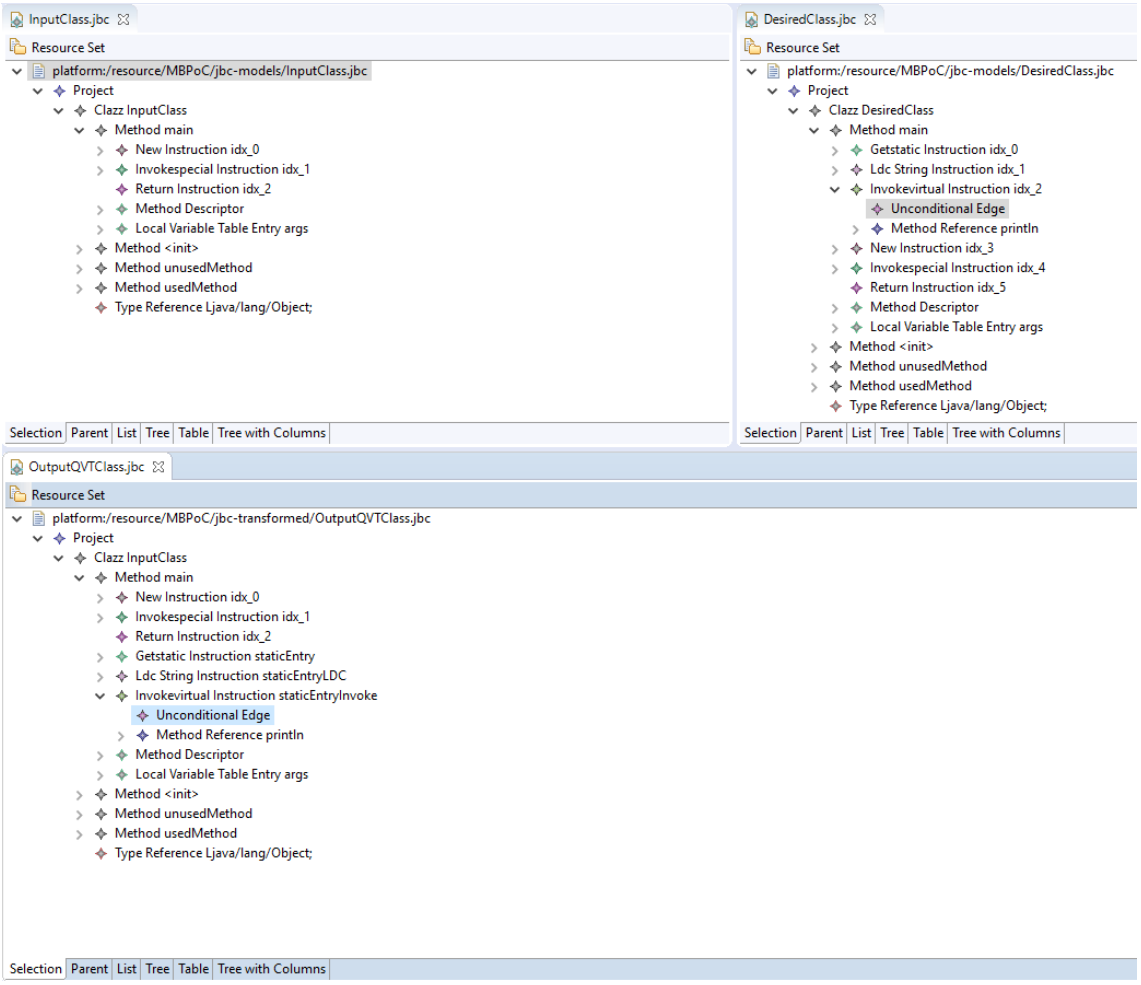


Figure 2: Output comparison

6 Conclusion

From the results of section 5 we can finally draw a conclusion to the question of "Which approaches currently exist for code-instrumentation and how do they compare?". It can be seen that code-instrumentation is used in a wide-variety of approaches and tools. Code-instrumentation is used from simple actions, such as logging each method, to more concise actions such as adding hooks to a method and allowing or developer-defined actions. Even though various forms of usage can be found, there exists a base set of required functionality of which most is present in modern libraries. A notable exception has been the absence of functionality that clearly handles set "hooks" and "probes" into the code. This behaviour can be found in various tools, ranging from adding probes for user-defined actions (BTrace 5.4.1) to adding probes for execution monitoring and code-coverage analysis (Clover 5.5.1, Cobertura 5.5.2 and JaCoCo 5.5.3). Usage of hooks can also be found in AOP tools in order to define where the weaving of code should take place (AspectJ 5.6.1, Compose* 5.6.2 and JAsCo 5.6.3).

While performing this research it was notable that code that makes use of a library for code-instrumentation seems to facilitate a better separation of concerns in the source-code itself. The tools that implemented their own functionality often had a mixed code-base with instrumentation functionality and other parts of the code, as is visible in the JFluid parts of VisualVM (5.3.1). The majority of the tools have opted to use one of the various existing frameworks, with modern tools more often opting for more modern frameworks. This may be because of various reasons, such as the ease-of-use or the support of additional functionality like stack-map generation and verification of the produced bytecode. Those more modern frameworks have been found to use patterns that are more similar to programming using a "normal" API, rather than having to program each bytecode instruction separately.

From the results of the PoC in section 5.9 it can be concluded that Mod-BEAM can indeed serve as a substitute for primitive concepts. It is quite obvious that the PoC itself is merely at the level of introducing logging to every method in a similar manner to SLF4J (5.4.2), but it should be noted that because this can be implemented, any of the other primitive concepts should also be possible to be implemented in practise.

7 Future Research

This research should form an understandable basis for prospective work involving code-instrumentation. There exists however plenty of room for more in-depth analysis. As defined in section 3.2, there exist various requirements that can be further investigated. Examples of this would be to further investigate on compleixty, redundancy between similar tools, redundancy between differing tools and solutions on how to resolve this redundancy. The PoC as mentioned in 4.3 and 5.9 is still extremely simple, and a more complex implementation could serve as an indicator of the usability of Mod-BEAM in order to help resolve issues regarding the investigated redundancy. This future work should then provide a complete and clear understanding of where any potential issues may lie and the possible viability of Mod-BEAM in order to solve these issues.

References

[1] A. de Roo, M. Hendriks, W. Havinga, and L. Bergmans. Compose*: a language-and platform-independent aspect compiler for composition filters.

[2] M. Dmitriev. Design of jfluid: A profiling technology and tool based on dynamic bytecode instrumentation. Technical report, Mountain View, CA, USA, 2003.

[3] Erik Hilsdale, Mik Kersten. Aspect-oriented programming with aspectj. included with the original source-code, tutorial.ppt.

[4] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM.

[5] R. D. Spenkelink. Porting compose* to the java platform. Master's thesis, University of Twente, 2007.

[6] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM, 2003.

A Proof of Concept

```
1  modeltype bytecode uses 'https://modbeam.bitbucket.io/jbc';
2  transformation QVTTransformation(in sourceModel : bytecode, out destModel : bytecode);
3
4  main() {
5      sourceModel.rootObjects().deepclone();
6      destModel.objects()[bytecode::Method]->map M2M();
7  }
8
9  mapping bytecode::Method::M2M() : bytecode::Method {
10     init {
11         result := self;
12     }
13
14     // The method name that should be used for printing
15     var printName: String := self.name;
16
17     // Store between which instructions the new instructions should be added
18     var startInstruction: bytecode::Instruction := self.firstInstruction;
19     if(self.name.equalsIgnoreCase('<init>')) {
20         printName := self._class.name + " constructor";
21
22         // In the case of a constructor
23         startInstruction := startInstruction.outEdges._end->first();
24     };
25     var nextInstruction: bytecode::Instruction := startInstruction.outEdges._end->first↩
           ();
26
27     // Create the actual print instructions
28     var staticInstr: bytecode::GetstaticInstruction := self.createPrintStatic();
29     var ldc: bytecode::LdcStringInstruction := self.createPrintLDC("IN: " + printName);
30     var invoke: bytecode::InvokevirtualInstruction := self.createPrintInvoke();
31
32     // A method may directly return if it is empty
33     if (startInstruction.oclIsTypeOf(bytecode::ReturnInstruction)) {
34         nextInstruction := startInstruction;
35         startInstruction := staticInstr;
36
37         firstInstruction := startInstruction;
38     };
39
40     // Link the instructions into a chain
41     startInstruction.outEdges->first()._end := staticInstr; // Instead of a new edge
42     self.linkInstructions(staticInstr, ldc);
43     self.linkInstructions(ldc, invoke);
44     self.linkInstructions(invoke, nextInstruction);
45 }
```

Listing 3: QVTo imperative transformation for the PoC (5.9) without helpers

Part II

Main Research

Chapter 2

Research Goal & Methodology

This chapter defines a set of requirements based on the conclusion of the research as can be found in Part I. From these requirements, a set of research questions is derived. These questions can be answered so that the requirements are met. Additionally, it is discussed how the answers to these questions can be measured.

2.1 Motivation

From the conclusion of our research in Part I, we now have a list of libraries, tools and concepts that are implemented by these tools. As our goal is to investigate the usability, reusability, advantages and drawbacks of Mod-BEAM, we can specify a list of things that we need to do. Since model-driven engineering requires a model-transformation language in order to function, we have to select which one is most suited for Mod-BEAM. As we have to prove that Mod-BEAM is usable, we can decide on a tool to reimplement. As we also have to verify the reusability, we pick a second tool to re-implement that has a shared primitive concept with the first tool we pick. The diverse ways in which the same concept is implemented should be sufficient proof that there is very little reusability among the original implementations. When we have picked multiple tools to implement, we also need to measure our implementation against the original in order to determine the complexity and quality of the implementation. For this, a set of measurement criteria have to be defined.

In the subsections below, a set of requirements is defined that concretely specify what we wish to achieve. The requirements can be separated into a main requirement and sub-requirements. The main requirement aims to assert a concrete goal that the end product should fulfill, while the sub-requirements should be the requirements that are needed to reach the goal. These requirements are realised through a set of research questions. The main research question should aim at fulfilling the goal requirement, while the sub-questions should aim to fulfill the sub-requirements.

2.1.1 Requirements

Main requirement

The research should create an overview of how existing code instrumentation approaches compare to the approach using model-transformations through Mod-BEAM, while also describing how the approaches compare and how this comparison can be fairly measured. As a result, this study should give a clear understanding of the viability of Mod-BEAM as a library for bytecode instrumentation.

Sub-requirements

R1 Provide an overview on the advantages and disadvantages of viable transformation languages that can be used with Mod-BEAM

R2 Provide an overview on how the implementation through Mod-BEAM can be measured

R3 Provide a set of tools for which the functionality will be reproduced

R4 Provide a set of specific concepts that will be reimplemented for each tool

R5 Provide an overview on the complexity of the implementations of the concepts in the original tools

R6 Provide an overview on the complexity of the implementations of the concepts in the new approach

R7 Identify if Mod-BEAM can be used to improve reusability in code instrumentation approaches

2.1.2 Research Questions

The main research question to be answered is:

What are the benefits and drawbacks when implementing code instrumentation as model transformation through Mod-BEAM when compared to the implementation as code transformation?

The sub-questions that should lead to the answer of the main question are:

- **In which aspects can Mod-BEAM be analysed in order to compare it to other approaches? (R1, R2)**

  The answer to this sub-question is required in order to determine exactly how to compare Mod-BEAM, and which aspects can be measured and compared.

- **What are the advantages and disadvantages of declarative or imperative transformations through Mod-BEAM and which languages are viable? (R1)**

  This sub-question is answered in order to determine the various advantages and disadvantages of various types of model-transformation languages. From this sub-question, the most suitable language(s) are used for the actual implementation. As there are multiple types of model-transformation languages, the final choice might impact the complexity of the final implementation of a concept.

- **Which concepts of which tools have potential reusability that cannot be utilised in the current implementation? (R3, R4)**

  A small-scale research is performed before actually delving straight into implementing code-instrumentation approaches. For this sub-question, the results of Part I are used to determine a proper starting point as to which tools or approaches could potentially share reusability.

- **How complex is a model-driven solution of instrumentation-based functionality for the viable tools and concepts? (R5, R6)**

  This sub-question uses the actual implementation and the results thereof and uses requirement R2 in order to determine the complexity of the original tool and the complexity of the tool as implemented as model-transformation. The measured data is required to later perform an actual comparison.

- **How does the model-driven implementation compare to the original implementation? (R7)**

  This sub-question compares the usability factors of the implementations with each-other. Usability factors include items such as complexity and other measurable criteria to determine where Mod-BEAM performs better or worse in various aspects.

- **How reusable is a model-driven solution using Mod-BEAM compared to the original implementation? (R7)**

  This sub-question compares the reusability factors of the implementation as compared to how this would be done with the original tools (if possible at all). This is yet another aspect for which Mod-BEAM can be considered superior or inferior, and together with the usability aspects lead up to an answer on the superiority or inferiority of Mod-BEAM.

2.2 Research Methodology

This research follows an iterative approach. Before this can be done, a primitive concept (a concept of instrumentation related functionality) to implement is picked from the results as found in the problem analysis of Part I. This document contains an overview of various tools, which instrumentation libraries these tools use and which primitive concepts are implemented by these tools. As such, a more in-depth look at the potential concepts that can be implemented as a model transformation is performed. The chosen concept should not be able to be reused in the way that it is currently implemented as a code transformation.

The first step in an iteration is to implement an existing tool of such a concept as a model transformation. The results created during this step can be analysed in order to compare the advantages and disadvantages of using a model transformation.

The second and final step in an iteration is the validation, during which we analyse quality aspects of the transformation code. These quality aspects can then be used to compare the final result with the original transformation, resulting in answers on the research questions as stated above. Various aspects of the *model-driven* code that can be validated are as follows:

- Lines of Code (LoC)

The amount of code that is related to the implemented concept serves as an easy estimate for how much work is required to implement that concept.

- Complexity

Complexity can be calculated through the Cyclomatic complexity (CYC). The smallest, largest and average CYC can be used to serve as an indicator of complexity which would influence aspects such as ease to create and extendibility.

- Modularity

The modularity of code should serve as an indirect indicator of how reusable the code within the transformation itself is. Modularity should take into account the size of the class and the cohesion of a class in order to come to a conclusion.

- Ease to create

Ease to create can be measured as the time one would need to create the functionality within the actual tool. This can be measured by taking the previously calculated modularity into account and estimating how hard it would be to work with the corresponding API.

- Extendibility

Extendibility is an estimate of how much time would be required to further develop the concept. This can depend on the desired functionality and the complexity of the existing code (lines, modularity).

Besides the analysis of the model-driven code, the original code-transformations are also compared in order to lead to a more complete understanding of the various advantages and disadvantages of each approach. Essentially, the same points as for the model-driven approach apply. To provide a fair comparison, the implemented concepts will have to make use of the original test suites (if available) of the original implementations. The original implementation of a concept and the new implementation as model-transformation should be proven to be functionally equal.

Chapter 3

Technical Background

This chapter contains the steps that are required to perform the practical section of this research. In order to realise a useful prototype, a set of research questions have to be answered beforehand. These research questions serve as the foundation to establish a prototype that is realistic, well-defined and measurable.

3.1 In which aspects can Mod-BEAM be analysed in order to compare it to other approaches?

An answer to this question can be found by taking various possible aspects of Mod-BEAM which can be defined as comparable. Core aspects that are interesting for this research are the usability, reusability and overall code quality of the implementations. These aspects can be further subdivided in order to make each aspect measurable. The final results of these various aspects will then lead to an overall conclusion on how our implementations compare to the original, and therefore also how practical usage of Mod-BEAM is.

3.1.1 Usability

Usability is an important, yet broad metric to define. There exist various ways to measure the usability of something, including software. Core aspects include measuring the usefulness, ease of use, ease of learning and user satisfaction for a product [1]. A problem with these criteria is that most of them are subjective. It is possible to extend these criteria by introducing some measurable aspects. This has already briefly been done in Section 2.2. Important aspects include the LoC, Complexity, Ease to Integrate, Extendibility and Modularity of the product. These aspects directly relate to the complexity of the implementation, and can be used as an indicator to measure how hard it is to work with the tool.

An issue that arises by using a model-transformation language is that they are often not supported by tools, as it differs from conventional programming. It is also not possible to reimplement the code transformation as native Java code either, as the functionality of mappings differs too much from Java code. As such, some metrics will need to be calculated manually. How this can be done is mentioned for each of the corresponding metrics.

Lines of Code

For the LoC, only the actual lines of the code are taken into account. This means that comments are completely ruled out, as they can be large or small depending on the granularity of explanation required for a class, but has little to do with the actual execution. The ease to integrate directly correlates to the complexity of how hard it is to make use of the output from the used tool. These statistics are calculated using Metrics Reloaded[1] (IntelliJ) for in-editor analytical information. Non-Commenting Lines of Code (NCLoC) can be manually calculated by removing all white lines and comments from the original file. Multi-line constructors should be moved to a single line. This can be done since the code-style remains the same between Java and our implementation as a model-transformation (see section 3.1.3 for information about code quality).

Complexity

Code complexity already has a very well defined way of being calculated, which is through CYC. CYC will be held into account in two manners for each class. The first type is the average CYC for each class. This average does not hold into account abstract functions or inherited methods. The second type is known as the weighted method complexity for each class. Similar to LoC, Metrics Reloaded can be to calculate the necessary data. In order to calculate the actual CYC (not the essential complexity or other variants) for the model-transformations, a simplified version of the McCabe CYC metrics is used[2].

Modularity

The modularity of the code serves as a decent indicator on the overall quality of the code. The modularity is calculated per class, according to the formulae derived by Emanuel et al. [2]. In order to calculate the modularity of a class, we require four metrics. These metrics are the following:

[1] https://plugins.jetbrains.com/plugin/93-metricsreloaded

[2] https://www.theserverside.com/feature/How-to-calculate-McCabe-cyclomatic-complexity-in-Java

- NCLoC
- Lack of Cohesion Method version 4 (LCOM4)
- CYC
- Number of functions F

The paper of Emanuel et al. [2] defines the LoC quality as follows (3.1 and 3.2):

$$LOC_Q = 0.0125xNCLoC + 0.375 for NCLoC <= 50 \tag{3.1}$$

$$LOC_Q = (NCLoC - 50)^{-2.046} for NCLoC > 50 \tag{3.2}$$

Function quality can be calculated as seen in equations 3.3 and 3.4.

$$F_Q = 0.172xF + 0.171 for F <= 5 \tag{3.3}$$

$$F_Q = (F - 4.83)^{-2.739} for F > 5 \tag{3.4}$$

Overall class quality can then be calculated as seen in equation 3.5.

$$C_Q = (LoC_Q + F_Q)/2xLCOM4 \tag{3.5}$$

Code cohesion is measured through LCOM4. LCOM4 indicates ”the degree of needed separation of classes into smaller classes” [2], which essentially means that any LCOM4 value above 1 means that the class can be split into multiple classes. An LCOM4 value of lower than 1 means that there are no functions in a normal (non-interface or abstract) class, while it does have variables. Finally, package level quality can be determined by taking $PQ = avg(CQ)$.

In this research, we calculate the LCOM4 metric by hand for each class. This can be done by deriving which variables are linked to which function. An alternative method of doing this is by drawing a graph from the class if it gets too complex[3]. We do this as, unfortunately, calculating the LCOM4 metric has become less popular over time due to claims of it being hard to correctly compute. This claim has been mentioned by one of the developers in the SonarSource team[4]. There was hope for jQana[5], but the download link leads to an outdated page. jPeek[6] would seem to be the most current tool, but we were unable to get the LCOM4 report generation to output sensible data.

[3]https://www.aivosto.com/project/help/pm-oo-cohesion.html

[4]https://stackoverflow.com/questions/25720061/how-to-get-lcomlack-of-cohesion-of-methods-metric-in-sonarqube-4-2

[5]https://github.com/cleuton/jqana/wiki/LCOM4-Calculation

[6]https://github.com/yegor256/jpeek

Ease to create

By taking into account the previously calculated modularity, we can create an estimate of how much effort it would take to create a new concept from scratch using the tool/API. We can take into account how high the total amount of NCLoC and CYC was when compared to the original implementation. From this, we can confirm which approach would give results with fewer lines of code and complexity.

Extendibility

Similar to the ease to create, we can make use of the NCLoC and CYC how much work it would be to add new functionality. Additionally, it can be reasoned which method would be easier to work with by taking the previously calculated modularity into account. A higher modularity or class quality would mean that it is easier to extend the product.

3.1.2 Reusability

Another code-concept for this research is the reusability of the created model transformations that perform code-instrumentation. Reusability has a lot in common with the overall usability of a piece of code. Higher readability and lower complexity can mean that code is reusable, but what we need to compare is not the existing reusability but rather the potential reusability or overall reusability of a certain concept. This is because our interest lies not in the reusability of functions in a normal object-oriented environment, but rather how a concept is implemented and how aspects can be reused by a similar concept in another tool. As such, we can use metrics such as the LoC and CYC, but also require an implementation that could potentially merge concepts from multiple tools. This is done by implementing specific tools using Mod-BEAM. If the conclusion is that the original implementation is hard to scale to incorporate similar functionality that other tools require, yet it is proven to be possible in Mod-BEAM, then that would indicate that Mod-BEAM holds an advantage in this aspect.

In essence, nearly everything can be created and made to be scalable using both model transformations or usual Java programming, but the importance here lies in the ease in which it can be done. If there is a need to add multiple passes or separate instructions, then that could be hard to implement using a Java Bytecode library, while model-transformations should, in theory, allow for clean and separable transformations.

3.1.3 Code quality

To ensure that the code is usable, maintainable and provides a fair comparison to the original implementation, the code has to be written as best as possible. While it might not be feasible to ensure that the code is perfect, there are guidelines that help verify that the code is decently written. The paper by Gerpheide et al. [3] provides a set of best practices that can be used to aid the quality of the model-transformation code. Model-transformation specific quality properties that were chosen to be upheld can be found in the list below:

- Few black-boxes,
- Few configuration properties
- Few input/output modules
- Few intermediate properties
- Few end sections
- Few mapping arguments
- Few when and where clauses
- Inheritance usage matches metamodel
- Little imperative programming (foreach loops)
- Little overloading
- Small init sections
- Few queries with side-effects

Besides this, some quality aspects similar to Object-oriented programming are also of concern. These are similarly mentioned by Gerpheide et al [3].

- Maintain detailed comments
- Consistent formatting
- Few dependencies
- Low code-duplication
- Few nested if-statements
- High usage of design patterns

- Interdependent functions near each-other
- Little (no) dead code
- Low CYC
- Minimal reassignment of objects
- Short function-chains
- Small function size
- Termination checked
- Low execution time (should be near-instant for reasonably sized classes)

Some quality aspects as mentioned by Gerpheide et al [3] are not implemented, these are:

- Pre- and post-conditions specified (not applicable)
- Small interfaces with other modules (not applicable)
- Small transformation size (not realistic)
- Confluence satisfied (not applicable)
- High test coverage (not applicable, usage of external tests and comparisons)
- More mappings than helpers (not realistic due to object creation reuse)
- More queries than helpers (not realistic)
- Deletion uses trashbin pattern (not applicable, the trashbin pattern is to assign objects to a parent object before removal [3])

Code formatting simply follows the Java syntax conventions. Classes use UpperCamelCase, while methods and properties use lowerCamelCase. Usage of constructors should be preferred when defining variables within a function. Mappings should be used to define the main transformation, while helper methods should be used to define reusable code (such as instruction batches) in order to increase reusability. Queries should be defined for complex retrieval actions. Usage of in/out variables should be avoided if possible as it reduces the readability of the code.

3.2 Model-transformation languages: declarative or imperative?

In order to create a library of model transformations, we first have to determine which transformation language to use. Transformation languages can be separated into three categories, namely declarative, imperative and hybrid. We limit our choices to transformation languages that make use of a textual syntax to define the rules of the transformation. While there exist multiple transformation languages for each category, there is no clear ”winner”, and nearly all languages are viable in most scenarios. As such, we simply list the advantages and disadvantages of each category and then proceed to pick a suitable transformation language from the chosen category or categories. In order to get started, a summary of the various categories is provided.

3.2.1 Declarative

A declarative language makes use of clear-cut rules that are executed once the transformation is triggered. These rules are defined on a specific level of the original metamodel and are executed for each corresponding element of the source model. A major advantage of declarative languages is that rules are clearly defined and executed automatically on a per-element basis. This means that for each model object matching a metamodel type, the defined rule is automatically found and executed. Moreover, a study has found that declarative languages are often faster than other categories of model-transformation languages [4]. Besides this, through the use of invariants, it is possible to perform computer-based analysis for declarative transformation languages [5]. This might provide merit for automated actions such as validation of completeness of the transformation without having to actually execute the code.

A declarative transformation language also has its disadvantages, however. The syntax differs from object-oriented programming and comes closer to query-languages, making it harder for some to adapt. Furthermore, instantiating new objects is often harder or might simply be impossible, as rules are more oriented towards changing one property towards another existing property. Furthermore, declarative languages are very strict on what can and cannot be done, which can either be advantageous or disadvantageous. It provides more strictness, making it more likely that the code is clean and readable. On the other hand, it would make it harder to define actions that follow a specific sequence.

3.2.2 Imperative

Imperative transformation languages often have a syntax that comes close to more ”usual” programming. The transformation is specified using a set of functions, which have to be called in order to be executed. Due to the proximity to usual programming, imperative languages are easy to understand and relatively easy to develop with. It does, however, also have its share of shortcomings. While in declarative languages mappings are always executed automatically, imperative languages require the developer to call these mappings sequentially. Due to its imperative nature, it is harder to perform computer-based analysis. Moreover, a study on declarative and imperative languages has found a positive correlation between execution time and called rules [4]. Some declarative languages tend to have a refinement mode in which the objects for which no rules have been defined are automatically copied to the target model, which is not present in the imperative languages that we have encountered to this point. A workaround for this is to perform a deep copy of all objects and work on the resulting objects, which leads to similar results as the refinement mode of ATL.

3.2.3 Hybrid

Hybrid languages consist of both declarative and imperative functionality. Declarative aspects can be used to define a set of rules, while imperative aspects can be used to define concrete methods. Hybrid languages possess most of the advantages from declarative and imperative languages, while also possessing some of the disadvantages. Since it is hybrid, the performance will be dependent on the amount of called rules, which means that it is slower than pure-declarative, but most likely faster than pure-imperative. The readability is high and because it is possible to make use of both declarative and imperative aspects, creating new objects is also fairly easy to do. Since it contains imperative aspects, it will likely be as hard to analyse for computers as pure-imperative code.

3.2.4 Choosing a language

Table 3.1 contains a list of both the advantages and disadvantages for each category as analysed earlier.

Based on the previous analysis, declarative languages win as the most performing and easy-to-analyse for computers. Imperative wins as the easiest to read and easy to use. Hybrid wins out as it is easy to read, create while potentially remaining fast. Since the initial proof-of-concept was created using Query/View/Transformation

Functionality	Declarative	Imperative	Hybrid
Easy to analyse by computers (e.g. using invariants)	O	X	X
Fast	O	X	O
Easy to read	O	O	O
Easy to create	X	O	O
Easy to refine	O	X	O
Control over flow of operations	X	O	O

Table 3.1: Concept summary of ASM

Operational (QVTo), this will also be used as the initial language for the initial prototype. After familiarizing with the concept of the chosen tools, a hybrid language will be used to develop the transformations instead. The requirements for the language candidates are very simple. There needs to be enough documentation to get started, and the language needs to be usable through Eclipse. As one of the most popular hybrid transformation languages is ETL [6], this was chosen to be the alternative to work towards. Since an imperative and declarative languages have a lot in common, migrating the language should not be too much work if everything works out as expected. At that point, we are able to compare whether QVTo or ETL is more viable to keep working with before continuing with the implementation of other tools.

3.3 Potential reusability

The results of our initial research as can be found in Part I brought forth enough information to decide on a valid entry-point for this research. One of the most foremost conclusions was that tools implementing Code-coverage analysis (CCA) such as Cobertura[7] and JaCoCo[8] both re-implemented the gathering and instrumenting of touch-points (or probe-points). Besides this, there was a noteworthy similarity between Cobertura eliminating ”duplicates” and JaCoCo eliminating ”consecutive duplicate frames” which has a high likelihood of being relevant to the usage of the ASM[9] instrumentation library of both tools. A potential extension of this functionality would be how Aspect Oriented Programming (AOP) languages make use of hook-

[7] http://cobertura.github.io/cobertura/
[8] https://www.eclemma.org/jacoco/
[9] https://asm.ow2.io/

or entry-points in order to determine where to weave the newly created code into the already existing code.

A criterion that the entry-point will need to fulfill is that it is not possible to reuse the functionality in the way that it is currently implemented. The implementations for both CCA tools are not consistent in the order that they execute operations, which would make it plausible that the code can be reused for one another. More definite is the fact that the code for aspect weaving cannot be used for the gathering of probe points.

To investigate the advantages and disadvantages of Mod-BEAM, we proceed by implementing the functionality of Cobertura. After this, we proceed by implementing the functionality of JaCoCo using the same base-transformation. This way we are able to gather analytical data on the usability as defined in Section 3.1.1. From this, we are also able to determine overlap between the implementations between the languages, besides being able to determine the various advantages and disadvantages between the implementations. The implementation of AspectJ remains hypothetical for now due to time constraints.

Chapter 4

First case study: Cobertura

This chapter contains information about the tool Cobertura and the reimplementation of the instrumentation logic of Cobertura. As reasoned in Section 3.3, this implementation aims to replicate the functionality of Cobertura known as touch-points. The implementation in this section is reused in the next section for the implementation of JaCoCo.

Touch-points are used to determine the coverage of classes, functions and statements after execution. Cobertura makes use of *ASM* in order to navigate the source code and process the instrumentation.

4.1 Cobertura

This section contains the analysis of the tool Cobertura. Data about the original implementation is gathered in order to later compare it to our implementation of the instrumentation logic.

4.1.1 Analysis

Cobertura requires a three-step process in order to create the desired results. In the first pass, an ASM specific issue is resolved. This is done by navigating through every statement in the code and keeping track of data in memory depending on the detected type. This data is contained in the class ”CodeFootstamp” which is used to represent an ASM ”footprint” in order to determine if code is nearly identical or different. This pass is relevant as it is part of the instrumentation chain, but might not be relevant for the implementation as model-transformation.

The issue that is resolved is that an if-statement in a ”finally” block of code would be generated three times, while it should only occur once, as noted in the comments

of the original source code[1]. The issue presents itself when try/catch blocks are required to be instrumented. This pass is separate from the other passes in the sense that there is no shared data in the Java code that stores IDs or other variables, unlike the other two passes. To see if we also have the same issue as the first pass, we can simply check if our results match the results of Cobertura when making use of try/catch blocks. More about this can be found in the proof of functional equality (Section 4.3).

Relevant classes and analytical data can be found below in Table 4.1. In order to correctly measure LCOM4, methods from the visitor pattern used through ASM are considered to be linked together as if they were called from the same function. Other data is gathered through Metrics Reloaded as mentioned in Section 3.1.1.

In the second pass of Cobertura's instrumentation process, the input code is traversed in order to build a ClassMap. This ClassMap object is then used in the third and last step to build a function that keeps track of all touch-points to be used in the actual coverage.

The data in the ClassMap is used in the third and final pass. This ClassMap stores several HashMaps:

- $\{$EventId, TouchPointDescriptor$\}$
- $\{$Label, TouchPointTouchPointDescriptor$\}$
- $\{$EventId, Label$\}$
- $\{$LineNumber, List $<$TouchPoint $>\}$

An EventID is a simple incrementing ID that is generated by Cobertura when visiting any kind of event. These EventIDs are used in order to access the TouchPoint data that was stored during the second pass. Instead, what Cobertura wants in the third pass, is to have an incremental CounterID for every TouchPoint. The EventIDs are unsuited for this, as there might be ”gaps” in the sequence of IDs, as events that are not considered to be a TouchPoint will still have a sequential ID generated for them (such as Labels).

By using ASM, not all line visits might be completely unique. For this, Cobertura waits until the second pass is completely finished before making a call to their ”assignCounterIds” function. This function makes use of every TouchPoint stored in the $\{$LineNumber, List $<$TouchPoint $>\}$ mapping. The functionality is quite straightforward, as it only increments a unique ID for each TouchPoint using an AtomicInteger idGenerator in order to avoid concurrency issues. One reason that these CounterIDs are not generated in the second pass might be because ASM works concurrently. While an AtomicInteger might resolve concurrency issues by locking and

[1] http://cobertura.github.io/cobertura/

incrementing sequentially, there is no guarantee that every event increments this AtomicInteger in the correct order due to race conditions.

In Java Bytecode, Labels can be used to keep track of specific addresses. This is then used so that any code-jumps can make use of the address that is assigned to a label. Cobertura makes use of Labels in order to determine if the destination of an event is a Jump event or not. In order to properly keep track of the desired CounterID, the usage of both the Label with TouchPointDescriptor mapping and the EventID with Label mapping is required. Relevant classes are shown in Table 4.2 below.

The third and last step also traverses through the entirety of the input code. The instrumentation code adds the instructions of the aforementioned ClassMap, as well as the counter instructions that are prepended and appended to the original instructions of the .class file.

The code in the third pass retrieves the TouchPoints stored in the second pass using the EventID as key. In this manner, the CounterID that was assigned for each TouchPoint can be used in order to create the appropriate instructions. All the code does during this pass is to create the appropriate instructions depending on the situation (line, JUMP, etc.).

Important to note is that Cobertura makes use of ASM, and ASM traverses the code in a code-flow oriented manner, which means that if a JUMP is performed, the code visits will traverse through this JUMP statement before continuing with the original code. An example would be how, with a simple visit as in Listing 4.1, the visited lines are not sequential. One would expect a sequence of lines as 70, 73, 77, 78, 79, 82, 84, 87 to be traversed in the same order as that it is listed. Instead, in ASM it could be traversed as 70, 78, 79, 82, 84, 77, 73, 87 depending on where branching occurs.

```
public void visitLineNumber(int line, Label label) {
        super.visitLineNumber(line, label);
        System.out.println(line);
}
```

Listing 4.1: ASM traversal

Functions relevant to the third pass can be found below in Table 4.3. Besides this, a number of classes are used in order to keep track of data. These can be found below in Table 4.4.

Class	LoC	Function count	Average CYC	Total CYC	LCOM4
CodeFootstamp	117	20	1.6	32	1
DetectDuplicatedCodeClassVisitor	35	4	1	4	1
DetectDuplicatedCodeMethodVisitor	166	17	2.25	36	1
DetectIgnoredCodeClassVisitor	47	5	1	5	1
DetectIgnoredCodeMethodVisitor	148	17	1.62	26	1

Table 4.1: Cobertura: First pass

Class	LoC	Function count	Average CYC	Total CYC	LCOM4
BuildClassMapVisitor	83	7	1.83	13	1
BuildClassMapTouchPointListener	42	9	1.00	9	1

Table 4.2: Cobertura: Second pass

Class	LoC	Function count	Average CYC	Total CYC	LCOM4
AbstractCodeProvider	178	10	2.11	19	1
AtomicArrayCodeProvider	98	5	1	5	1
FastArrayCodeProvider	71	5	1	5	1
InjectCodeClassInstrumenter	102	6	2.25	9	1
InjectCodeTouchPointListener	98	10	1.8	18	1

Table 4.3: Cobertura: Third pass

Class	LoC	Function count	Average CYC	Total CYC	LCOM4
ClassMap	361	22	3.05	67	1
JumpTouchPointDescriptor	62	6	1	6	1
LineTouchPointDescriptor	68	5	1	5	1
SwitchTouchPointDescriptor	109	8	1.12	9	1
TouchPointDescriptor	78	6	1	5	1
FindTouchPointsMethodAdapter	306	19	1.58	30	1

Table 4.4: Cobertura: Misc

4.1.2 Instrumentation

While we have created various metrics of the Cobertura source code, that does not yet provide useful information on how to reproduce the results of the actual implementation. For this we need to know the following:

- Under which conditions are instructions generated
- Which instructions are generated under the aforementioned conditions

The instrumentation part of the implementation of touch-points in Cobertura can be separated into two parts. The first part is the counter which contains the information if a function was executed after running or not, and which branches were executed. The second part is the set of "__cobertura" functions that are used afterwards to use the execution data in order to calculate the actual coverage metrics. The actual execution of these functions is no longer part of the instrumentation but rather the analysis itself.

In order to realise this, we have two sources. The first source is the original source code of Cobertura. For this, we need to replicate the instructions as performed through ASM for our model-transformation implementation. The second source is created by letting Cobertura instrument a set of files and checking what exactly was inserted, detecting patterns and decompiling if required for a better overview.

The code in Listing 4.2 shows an extremely simple Java constructor. This will be the input for our simple example. After instrumenting the file with Cobertura, it can be decompiler back to a Java source file. This is done using CFR. The output for this is visible in Listing 4.3.

```
public InputClass() {
        System.out.println("Default␣constructor");
}
```

Listing 4.2: InputClass.java (input)

```
public InputClass() {
    InputClass.__cobertura_init();
    int n = 0;
    int[] arrn = __cobertura_counters;
    arrn[1] = arrn[1] + 1;
    int[] arrn2 = __cobertura_counters;
    arrn2[2] = arrn2[2] + 1;
    System.out.println("Default␣constructor");
    int[] arrn3 = __cobertura_counters;
    arrn3[3] = arrn3[3] + 1;
}
```

Listing 4.3: InputClass.java (output)

Figure 4.1 below shows that the actual bytecode (as represented by Mod-BEAM) of this file contains a decent amount of instructions, for what seems like a small piece of code. Cobertura quickly adds a lot of additional instructions to this. The figure depicts the init function which essentially originally only had a small set of instructions: Aload, Invokespecial, Getstatic, LdcString, Invokevirtual and Return, as it is the default constructor of the class with a simple print statement. For this, Cobertura adds a set of instructions per-method (shown in yellow) and per-line (shown in blue). Per-method, an InvokeStatic, LDC and IStore are added to set up the measurement data. The Getstatic (idx3) to IAStore (idx9) measure the actions of the single line.

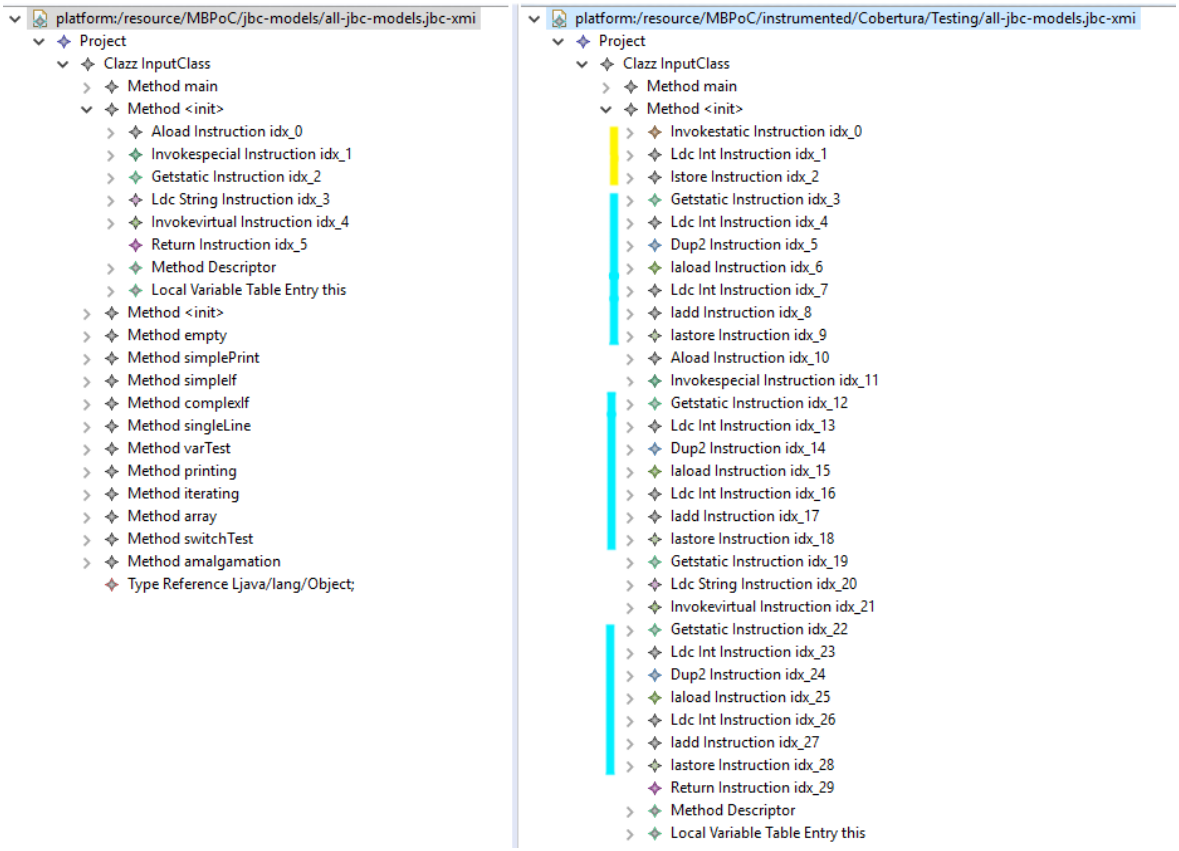


Figure 4.1: Uninstrumented (left) vs Instrumented (right)

4.1.3 Instructions in detail

What was shown in Figure 4.1 was only the init function with a single line of code. Notable is that Cobertura adds sets or ”batches” of instructions to every individual line of instructions. The original source code of Cobertura is important for this as it shows us what the trigger is for a specific batch of instructions. The actual instrumented code as viewable through the Mod-BEAM toolset shows us what we actually need to work towards, while also presenting the ability to inspect the final instruction

traversal order without presenting too much information at once (albeit still a lot). Instruction batches from Cobertura can be prepended to the first instruction or appended to the last instruction depending on the type. Figure 4.2 shows our analysis of the desired results.

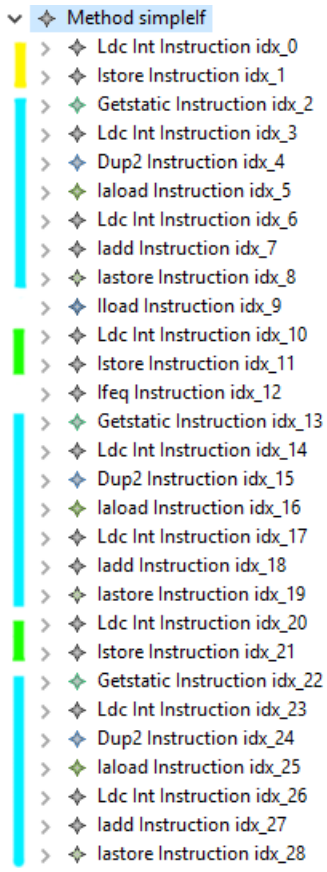


Figure 4.2: Instrumentation analysis

Figure 4.2 depicts several types of batches as instrumented by Cobertura. Shown in white are the original instructions that were also present in the original .class file. Yellow lines are lines added for each method. Blue lines are prepended to each first instruction on a new line. Shown in green are instructions that are only placed before an actual if-statement, or after a batch that originates from an if-statement. The instructions visible in the image are only a small part of the entire instrumented method.

The added instructions can be summarised as follows:

- Before each first instruction on a line
- Before each if-instruction
- After each if-instruction

  - Appended instructions for the 'false' branch
  - Appended instructions for the 'true' (GOTO) branch

- Before each switch-instruction
- After each switch-instruction
  - Appended instructions for each switch-branch

Each batch of instructions has parameters so that they are considered to be on the same line as the original instructions. What each batch of instructions does is to increment a number in an array of numbers. Exceptional statements such as the if-statement have additional instructions that either reset the local counter or increment an already existing index further. As GOTOs are also considered as an instruction, the default 'first instruction of a line' batch is also added before the instruction position. The GOTO has a few additional instructions that reset the current counter. The original functions by Cobertura can be used as a guideline to which batches of instructions we would actually need. These functions are:

- generateCodeThatSetsJumpCounterIdVariable
- generateCodeThatZeroJumpCounterIdVariable
- generateCodeThatIncrementsCoberturaCounterFromInternalVariable
- generateCodeThatIncrementsCoberturaCounter
- generateCodeThatIncrementsCoberturaCounterIfVariableEqualsAndCleanVariable

The above items can serve as an indicator of which functions are needed in our implementation.

4.2 Cobertura as model transformation

This subsection contains information about the implementation of Cobertura's instrumentation passes as a model-transformation using Mod-BEAM.

4.2.1 Passes

One of the key points in the instrumentation is that we want the same results as Cobertura, but we do not necessarily want the same process. As mentioned earlier, Cobertura makes use of 3 individual passes that each traverse all elements. The first pass detects duplicates in order to handle ASM specific bugs, which can be skipped in our implementation. The second pass creates an internal ClassMap object in order to store a sequence of touch-points. After the second pass, CounterIDs are generated for each TouchPoint. The last pass does the actual instrumentation, making use of the IDs stored in the ClassMap. Using a model-transformation, however, we have all the necessary information in a central place. As such, we can simply keep track of necessary IDs while directly adding the necessary instructions. These IDs can then later be used in order to generate a separate function. There are no concurrency issues to keep track of, which means that the code can be kept straightforward and simple. The result of this is that the model-transformation directly executes what is known as the third pass in the implementation of Cobertura. The third pass directly creates the CounterIDs in sequence while also adding the instructions for each line or JUMP statement. While visiting each line, we still have to store the TouchPoints in an array in order to add the TouchPoints in a line number-ordered manner in one of the static Cobertura functions ("__cobertura_classmap0"). This means that only the saved TouchPoints need to be traversed once more when generating these line number-ordered SequenceIDs.

4.2.2 Traversal and control-flow

First and foremost, we need to have a consistent way to traverse through the codebase. This can easily be done by making use of mappings in the code-transformation language. As we are using QVTo for our first implementation, we can start by walking through each instruction by creating an instruction-level mapping. Cobertura walks through code on a line-based fashion, which means that each set of instructions separated by a newline is considered a new set of instructions to be instrumented.

As the Mod-BEAM metamodel makes use of edges in order to handle the control-flow of the code, we are able to check if the next line is on a different line than our current line, which would could be used in order to insert instructions between the

last instruction of a line and the first instruction of a new line. Simplified usage of outgoing edges can be seen in listing 4.4.

```
mapping Instruction::I2I(): Instruction {
    // Default lines only have one outgoing edge.
    // We can simply check if the first edge is on a different line.
    if(self.outEdges != null && self.outEdges->first().linenumber != linenumber) {
        // Insert instructions that correspond
        // with the linenumber of self.outEdges->first.
        // Link outgoing edge to newly added first instruction.
    }
}
```

Listing 4.4: transformation_outgoing.qvto

While it is possible to reason this way, what we want to replicate is adding the instructions before each first instruction on a line. The aforementioned method would be hard to use when also taking if statements and GOTOs into account in the same manner that Cobertura does this. A separate but hard-to-resolve issue is the fact that other instructions (if/GOTO) could indirectly lead to the target of an outgoing instruction. This would require keeping track of the modified instructions in order to then navigate through all instructions again and resolve outdated edges. To resolve this issue, an easier alternative has been implemented: to allow access for incoming edges besides outgoing edges. As is visible in listing 4.5, this not only made it easier to reason about, but it also allows us to redirect all incoming edges to a new instruction through a simple helper method.

```
mapping Instruction::I2I(): Instruction {
    // If it is the target of a if/GOTO then it is always
    // the first instruction of a line.
    // This is also the case if the previous instruction
    // was the last instruction on a line.
        if(inEdges.start->first().linenumber != linenumber) {
        // Proceed by inserting new instructions.
        // Link all incoming edges to the new first instruction.
    }
}

// Used to link every incoming end to a new instruction.
helper Instruction::linkEnd(instruction: Instruction) {
        // Attach every end to this.
        self.inEdges->forEach(edge) {
                edge._end := instruction;
        }
}
```

Listing 4.5: transformation_incoming.qvto

4.2.3 Variable index

As Cobertura requires the use of a variable that is purely defined through code-instrumentation, it also needs to make use of the correct variable index. In the results of Cobertura's instrumentation, it is visible that the index is inserted before the first original index. This means that if originally a variable with index 1 existed, the instructions accessing this index would now use index 2, while the newly added "__cobertura_counter" would use varIndex 1 instead. After some research, it turned out that this variable likely originates from ASM automatically computing the variable indices. Since manual insertion and moving around of variables was more likely to cause issues rather than resolve them, an easier implementation would be to, as long as indices are not computed automatically, simply append the index of the new variable to the maximum variable index instead of inserting it. Yet another problem with this is that we would first need to traverse all possible usages of a varIndex in order to compute the maximum index so that we can use the maximum index + 1. The solution for this is to simply use an arbitrary high index number that is currently unused. While it might sound like an ugly solution, it is by far the easiest while remaining clean. An arbitrary high index such as 999 would mean that there is enough space for 998 index slots to be used before that point. The code runs as one would expect, and when decompiling the generated .class file back to a .java file using either CFR or JDCore it can be seen that the variable is properly generated without causing any conflicts or issues.

4.2.4 ID Ordering

As shortly mentioned in subsection 4.1.1, the usage of ASM makes Cobertura traverse the code-base in a code-flow based manner. This means that any if statement or GOTO can cause the traversal order to be changed. On the other hand, using a model-transformation language, we sequentially traverse the code. This means that we start at the first line number, and progress for each line that has instructions until we reach the end of the file. The main difference that this makes is that Cobertura adds counterIds in their control-flow order, which means that our results differ in this aspect when compared to the original result. Besides this, matters such as if- or switch-cases make this issue even more complicated. The location at which a batch of instructions is added (after switch or after GOTO resume) can further complicate the resulting ID order. This is because an if statement has a corresponding GOTO, but a GOTO might be able to exist without originating from an if statement (e.g. because of iterators).

It can be argued that if ASM made use of a sequential traversal order that they would simply make use of the IDs generated in that manner for analysis instead of

the current ID order. It does remain possible to mimic this flow of actions through a model-transformation language, but it would make the code more complex. AS such, instead of fully implementing it, a description of the implementation is provided instead:

- Iterate through all instructions
- If an instruction is an IF statement, recursively call this method for each outgoing edge, creating a new ”path”
- If an instruction is a GOTO statement, break off the current ”path” to avoid iterating endlessly

The above solution works for simple cases but might break when there are more GOTOs generated than there are if/else branches. As such, it remains to be tested if it works using iterations, but it does work in code without them. It should, therefore, be possible to create a full copy of the original Cobertura ordering. As mentioned earlier, the counterargument is that if an application would be created from scratch (for example using Mod-BEAM), it would make more sense to go with the native ordering sequence.

4.2.5 Code quality

As detailed in Section 3.1.3, a set of code-quality measures were taken into account. One of the more notable aspects of creating a properly readable transformation was to limit the usage of in/out keywords in functions. Instead of modifying parameters by reference we chose to make use of intermediate classes that could be used to store multiple values. This way, instead of passing an edge to a helper that creates instructions in order to link them, we were able to simply return the first and last instruction of a batch and modify the control-flow in the mapping itself. Other quality aspects are fairly obvious and were not difficult to implement. Something that quickly became notable was that the implementation could be separated into multiple parts. The first part containing constructors and some global functions could be reused for any kind of implementation that makes use of QVTo, and could potentially be placed into a separate transformation that could then be extended. The second part would then be the implementation itself which is specific to the instrumentation of Cobertura logic. Not separating the mapping of instructions into multiple parts was a deliberate choice, as splitting one mapping of instructions into lines, if statements, switch cases and GOTOs with WHEN clauses would cause the ID ordering to be even more different than it is already. This is because the mappings are not executed in parallel and have to be called sequentially due to the imperative nature

of QVTo. Alternatively, the function could be placed in separate helper methods, but it is unknown just how big the performance impact of this would be. As such, it is left as it is for now.

There still exists room for improvement in the total lines of code of the implementation. Linking instructions could potentially be done automatically by using an insert function rather than linking everything afterwards. Another example would be how the constructors take a lot of space as there is no overloading. Some constructors could also be smarter by generating a reference based on the input, rather than having to generate an object separately.

4.3 Assessing functional equality

In order to able to compare our implementation of the instrumentation logic from Cobertura, we will first have to validate that the functionality remains the same. As briefly mentioned in Section 2.2, we aim to do this by reusing test-data from Cobertura. Due to the various minor differences such as ordering, we are unable to directly plug our implementation into the original test-suite of Cobertura. As an alternative, we have chosen to create a large test class, as can be found in Appendix A. This test class consists of functionality from various test cases, expanded upon with some more complex scenarios such as inner loops in order to ensure that most situations are handled correctly.

To compare our implementation, we first have to verify that it has no impact on the original code. This is important because we add variables in a way that is different from the norm and is not yet automatically computed. Our instructions should not be in the way of the original, regardless of what we do. To test this, we added a number of print statements that indicate how many times functions were iterated and which values were printed. The output of our instrumented class was then compared with the unmodified version. The results of this can be seen in Figure 4.3.

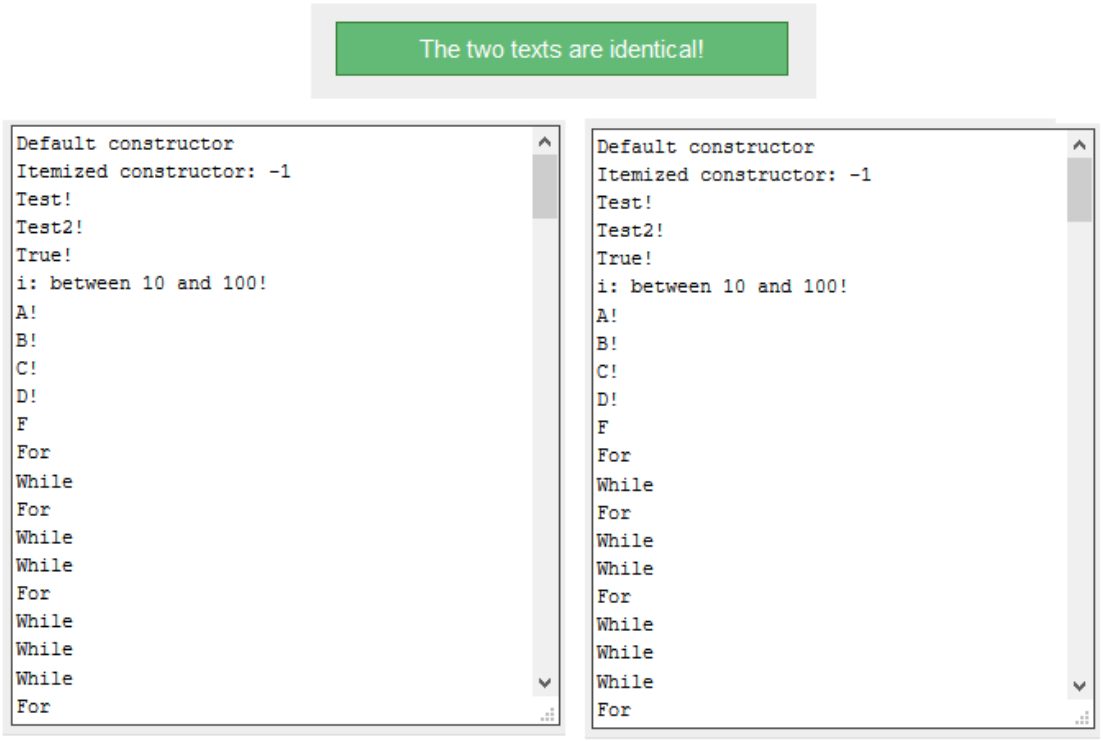


Figure 4.3: Test output comparison

Now that we have ensured that the original functionality remains the same, we can proceed by comparing our output to the output generated by CObertura. Instead of comparing raw data such as class files, it would be easier to find a decent decompiler and compare the decompiled results. The decompiled results can then be processed through a text comparison in order to find where the differences lie. The decompiler used CFR[2] as the results are readable and the decompiler is up-to-date. We first start by comparing the differences in the instrumentation batches. A sample of the resulting output can be seen in Figure 4.4.

[2]https://www.benf.org/other/cfr/

```
306    private void array() {
307        var1_1 = 0;
308        v0 = InputClass.__cobertura_counters;
309        v0[70] = v0[70] + 1;
310        v1 = new int[3];
311        v2 = InputClass.__cobertura_counters;
312        v2[72] = v2[72] + 1;
313        v1[0] = 5;
314        v1[1] = 10;
315        v1[2] = 30;
316        v3 = InputClass.__cobertura_counters;
317        v3[71] = v3[71] + 1;
318        items = v1;
319        v4 = InputClass.__cobertura_counters;
320        v4[73] = v4[73] + 1;
321        var3_3 = items;
322        var4_4 = var3_3.length;
323        var5_5 = 0;
324        ** GOTO lbl30
325 lbl-1000: // 1 sources:
326        {
327            v5 = InputClass.__cobertura_counters;
328            v6 = var1_1;
329            v5[v6] = v5[v6] + 1;
330            var1_1 = 0;
331            item = var3_3[var5_5];
332            v7 = InputClass.__cobertura_counters;
333            v7[77] = v7[77] + 1;
334            System.out.println(item);
335            v8 = InputClass.__cobertura_counters;
336            v8[74] = v8[74] + 1;
337            ++var5_5;
338 lbl30: // 2 sources:
339            var1_1 = 76;
340            ** while (var5_5 < var4_4)
341        }
342 lbl32: // 1 sources:
343        v9 = InputClass.__cobertura_counters;
344        v9[75] = v9[75] + 1;
345        var1_1 = 0;
346        v10 = InputClass.__cobertura_counters;
347        v10[78] = v10[78] + 1;
348    }
349
350    private void switchTest(int n) {
351        void param;
352        int n2 = 0;
353        int[] arrn = __cobertura_counters;
354        arrn[79] = arrn[79] + 1;
355        n2 = 80;
356        switch (param) {
357            case 1: {
358                if (80 == n2) {
359                    int[] arrn2 = __cobertura_counters;
360                    arrn2[82] = arrn2[82] + 1;
361                    n2 = 0;
362                }
363                int[] arrn3 = __cobertura_counters;
364                arrn3[87] = arrn3[87] + 1;
365                System.out.println("Switch 1");
366                int[] arrn4 = __cobertura_counters;
367                arrn4[88] = arrn4[88] + 1;
368                break;
369            }
```

```
304    private void array() {
305        var999_1 = 0;
306        v0 = InputClass.__cobertura_counters;
307        v0[70] = v0[70] + 1;
308        v1 = new int[3];
309        v2 = InputClass.__cobertura_counters;
310        v2[71] = v2[71] + 1;
311        v1[0] = 5;
312        v1[1] = 10;
313        v1[2] = 30;
314        v3 = InputClass.__cobertura_counters;
315        v3[72] = v3[72] + 1;
316        var1_2 = v1;
317        v4 = InputClass.__cobertura_counters;
318        v4[73] = v4[73] + 1;
319        var5_3 = var1_2;
320        var4_4 = var5_3.length;
321        item = 0;
322        ** GOTO lbl30
323 lbl-1000: // 1 sources:
324        {
325            v5 = InputClass.__cobertura_counters;
326            v6 = var999_1;
327            v5[v6] = v5[v6] + 1;
328            var999_1 = 0;
329            items = var5_3[item];
330            v7 = InputClass.__cobertura_counters;
331            v7[74] = v7[74] + 1;
332            System.out.println(items);
333            v8 = InputClass.__cobertura_counters;
334            v8[75] = v8[75] + 1;
335            ++item;
336 lbl30: // 2 sources:
337            var999_1 = 77;
338            ** while (item < var4_4)
339        }
340 lbl32: // 1 sources:
341        v9 = InputClass.__cobertura_counters;
342        v9[76] = v9[76] + 1;
343        var999_1 = 0;
344        v10 = InputClass.__cobertura_counters;
345        v10[78] = v10[78] + 1;
346    }
347
348    private void switchTest(int n) {

349        int n2 = 0;
350        int[] arrn = __cobertura_counters;
351        arrn[79] = arrn[79] + 1;
352        n2 = 80;
353        switch (n) {
354            case 1: {
355                if (80 == n2) {
356                    int[] arrn2 = __cobertura_counters;
357                    arrn2[81] = arrn2[81] + 1;
358                    n2 = 0;
359                }
360                int[] arrn3 = __cobertura_counters;
361                arrn3[82] = arrn3[82] + 1;
362                System.out.println("Switch 1");
363                int[] arrn4 = __cobertura_counters;
364                arrn4[87] = arrn4[87] + 1;
365                break;
366            }
```

Figure 4.4: Instruction output comparison

As is visible in the image, the majority of the changes are from how variables and IDs are handled. These issues were already covered in Section 4.2.3 and Section 4.2.4. The output of the TouchPoints is visible in Figure 4.5.

```
putLineTouchPoint(93, 72, "array", "()V");
putLineTouchPoint(96, 73, "array", "()V");
putLineTouchPoint(96, 74, "array", "()V");
putJumpTouchPoint(96, 76, 75);
putLineTouchPoint(97, 77, "array", "()V");
putLineTouchPoint(99, 78, "array", "()V");
putLineTouchPoint(102, 79, "switchTest", "(I)V");
putSwitchTouchPoint(102, Integer.MAX_VALUE, new int[]{82, 86, 85}
putLineTouchPoint(105, 87, "switchTest", "(I)V");
putLineTouchPoint(106, 88, "switchTest", "(I)V");
putLineTouchPoint(108, 89, "switchTest", "(I)V");
putLineTouchPoint(109, 90, "switchTest", "(I)V");
putLineTouchPoint(111, 91, "switchTest", "(I)V");
putLineTouchPoint(114, 92, "switchTest", "(I)V");
putLineTouchPoint(118, 93, "amalgamation", "()V");
putLineTouchPoint(119, 94, "amalgamation", "()V");
putLineTouchPoint(120, 95, "amalgamation", "()V");
putLineTouchPoint(121, 96, "amalgamation", "()V");
putLineTouchPoint(122, 97, "amalgamation", "()V");
putLineTouchPoint(124, 98, "amalgamation", "()V");
putJumpTouchPoint(124, 100, 99);
putLineTouchPoint(125, 101, "amalgamation", "()V");
```

```
putLineTouchPoint(93, 72, "array", "()V");
putLineTouchPoint(96, 73, "array", "()V");
putLineTouchPoint(96, 74, "array", "()V");
putJumpTouchPoint(96, 76, 75);
putLineTouchPoint(97, 77, "array", "()V");
putLineTouchPoint(99, 78, "array", "()V");
putLineTouchPoint(102, 79, "switchTest", "(I)V");
putSwitchTouchPoint(102, Integer.MAX_VALUE, new int[]{81, 83, 85});
putLineTouchPoint(105, 87, "switchTest", "(I)V");
putLineTouchPoint(106, 88, "switchTest", "(I)V");
putLineTouchPoint(108, 89, "switchTest", "(I)V");
putLineTouchPoint(109, 90, "switchTest", "(I)V");
putLineTouchPoint(111, 91, "switchTest", "(I)V");
putLineTouchPoint(114, 92, "switchTest", "(I)V");
putLineTouchPoint(118, 93, "amalgamation", "()V");
putLineTouchPoint(119, 94, "amalgamation", "()V");
putLineTouchPoint(120, 95, "amalgamation", "()V");
putLineTouchPoint(121, 96, "amalgamation", "()V");
putLineTouchPoint(122, 97, "amalgamation", "()V");
putLineTouchPoint(124, 98, "amalgamation", "()V");
putJumpTouchPoint(124, 100, 99);
putLineTouchPoint(125, 101, "amalgamation", "()V");
```

Figure 4.5: TouchPoint output comparison

As is visible in the figure above, the only differences are in the IDs generated for switch-cases. This is because the IDs for TouchPoints are labelled in a sequential order based on line number rather than the original IDs, the only exception being that switch-cases require a reference to multiple original event IDs.

There are still a few issues that remain in the implementation, and there might be more as the test-cases cannot identify all edge-cases. The first remaining issue is that multi-condition statements do not result in the same amount of instructions as the implementation by Cobertura. This is because of an early implementation choice to mimic the ID generation sequence as close as possible. As such, for if statements, part of the instructions are generated when the if instruction is visited and another part is generated when the corresponding GOTO instruction is visited. The issue can be resolved by moving the transformation logic from the GOTO to the if statement and making some adjustments to the batch-creation rules. This has been completed in a new revision of the implementation, but even with the changed rules, it was proven to be extremely difficult to obtain identical results. This is because the resulting bytecode is complex even with simple examples. In the latest version of our implementation, only a few minor differences remain with the original implementation from Cobertura, even when encountering multi-condition statements.

Another difference is the processing of try/catch/finally blocks. While it is now supported by Mod-BEAM, our implementation does not yet handle edges of the type ”exception” correctly. Resolving this is also possible by implementing the correct logic, but we believe that our goal to proof that implementing the base functionality of Cobertura has sufficiently been reached, as most functionality is already implemented.

4.4 Migration to ETL

During the attempt to migrate from QVTo to ETL, both the advantages an disadvantages of using QVTo came to light. An advantage of using ETL over QVTo is the ability to make use of declarative mappings instead of imperative mappings. This would allow us to split the instruction mapping into several mappings, such as a mapping for if statements or switch cases, without impacting the order of ID generation. Making use of declarative mapping would also serve to spread code complexity and the lines of code across various functions rather than keeping a lot of logic in a single function. Another advantage would be that with declarative mappings it would be easier to perform things that should be done after each method or class, such as resetting a counter. This can still be done in QVTo but would be a bit harder and likely slower due to the necessary checks.

During the migration, we also found several disadvantages of migrating from QVTo to ETL. The first disadvantage we encountered is that ETL lacks examples. An overview of the functionality could only be found in the Epsilon book[3], with the content about ETL itself being extremely small. One of the primary resources used to gain a proper understanding of the various functions in QVTo would be one of the presentations of EclipseCon2008 [7] and EclipseCon2009 [8], while also relying on questions posted on the forums. For ETL it turned out to be difficult to even find how to keep all objects without defining a rule for each object type, similar to ATL's ”refinement” mode.

While QVTo supports constructors, ETL does not. An alternative for this would be to use ETL's ”operations” in order to create objects, such as a ”makeReturnInstruction()” operation, but this would seem like a misuse of the functionality. The functionality to use constructors turned out to be a necessity for our transformation, as without using them we would have to define a name, line number, reference or other data for each instruction that we generate. Not doing this would bloat the amount of code with several hundreds of NCLoC. Unfortunately, functionality that is missing from QVTo is constructor overloading, leading to an unnecessarily large amount of code dedicated to setting basic class fields.

The final disadvantage would be the extent to which editor and programming support have been implemented. Figure 4.6 shows that for ETL there are no suggestions based on the metamodel that is currently being used. Errors are only generated upon execution. Figure 4.7 shows that for QVTo suggestions are displayed for the object that was created. Syntactical errors prevent the code from executing and the editor itself displays on which line something is incorrect. Creation of objects that do not exist or making use of constructors that are not defined also display an

[3] https://www.eclipse.org/epsilon/doc/book/

error in the editor itself.

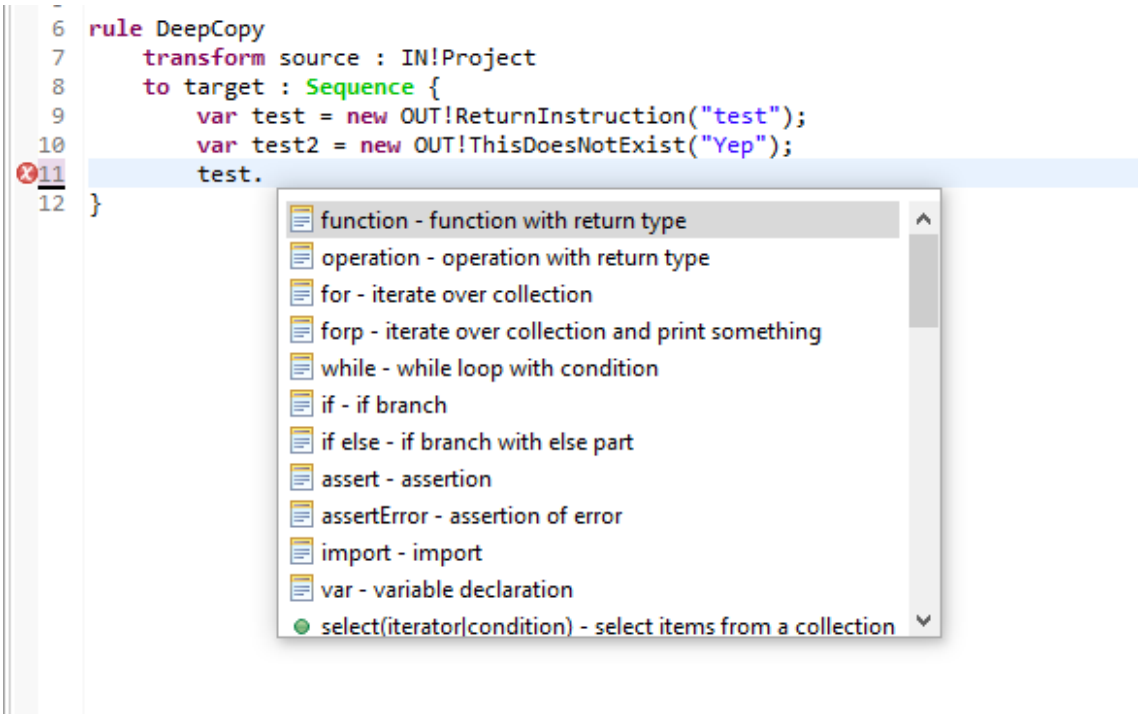


Figure 4.6: ETL suggestions

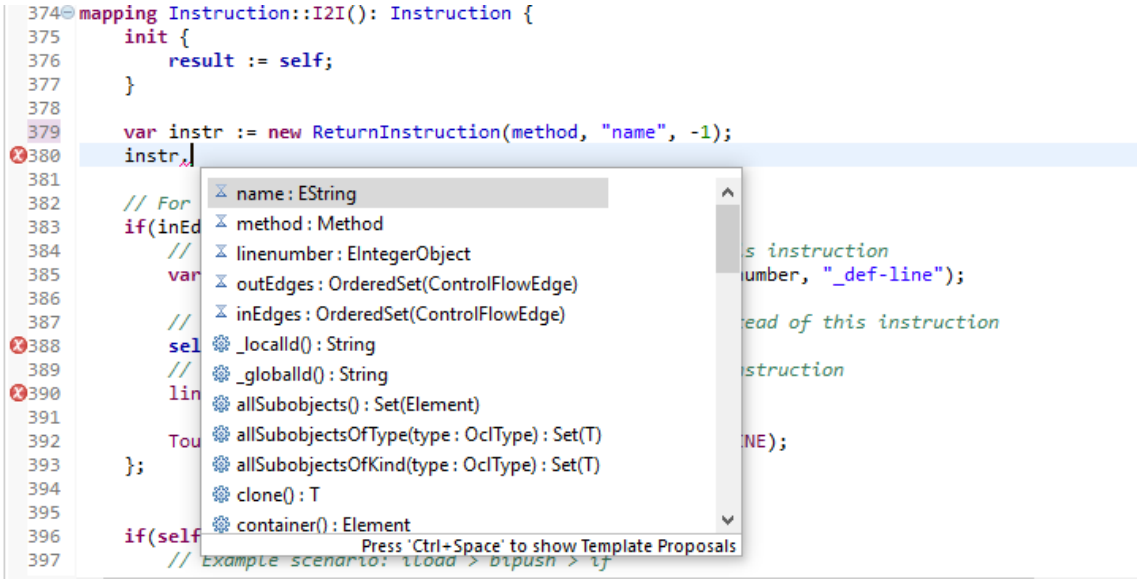


Figure 4.7: QVTo suggestions

These results were enough for us to decide to keep using QVTo for our current research. Others might find the advantages that ETL provides with declarative mappings more appealing, but QVTo is more developed in terms of usability.

4.5 Reimplementation analysis

In order to compare the quality of Cobertura and our implementation using MoDBEAM and QVTo, we first have to create the necessary raw data as mentioned in Section 3.1.1. We will need to calculate the NCLoC, total CYC and LCOM4 values. The lines of code, as determined by trimming empty lines and comments, amount to a total of 599 lines. Of those lines, 190 lines are used for setup, constructions and API-like functions. The other 409 lines of code are used for the implementation itself. The CYC is calculated per function or group of functions. Table 4.5 contains the CYC data for the constructors and API. Table 4.6 contains the CYC data for the instruction batch pass. Lastly, table 4.7 contains the CYC data for the "__cobertura" functions.

Class	**Total CYC**
Main	1
Helpers (API, 2)	2
Constructors	28
Total	31

Table 4.5: Cobertura as model transformation: setup and API

Class	**Total CYC**
Global constructors	2
Constructors	1
Helper linkEnd(I):	2
Helper LinkEnd(I, I)	5
Mapping C2C	1
Mapping M2M	6
Mapping I2I	17
Batch preIf	1
Batch postIf	1
Batch switch	1
Batch goto	1
Batch default	1
Total	39

Table 4.6: Cobertura as model transformation: main functions

Class	**Total CYC**
Mapping C2Cobertura	1
Helper coberturaInit	1
Helper classMap0	7
Helper classMap	1
Helper getAndResetCounters	1
Total	11

Table 4.7: Cobertura as model transformation: special functions

This amounts to a total of a CYC of 81 over a total of 599 lines of code, spread over 29 constructors, 15 helpers and 4 mappings. The final remaining variable is LCOM4, which, as is quite obvious, results in 1.0 as can be seen in Figure 4.8. Circles represent variables, while rectangles represent functions.

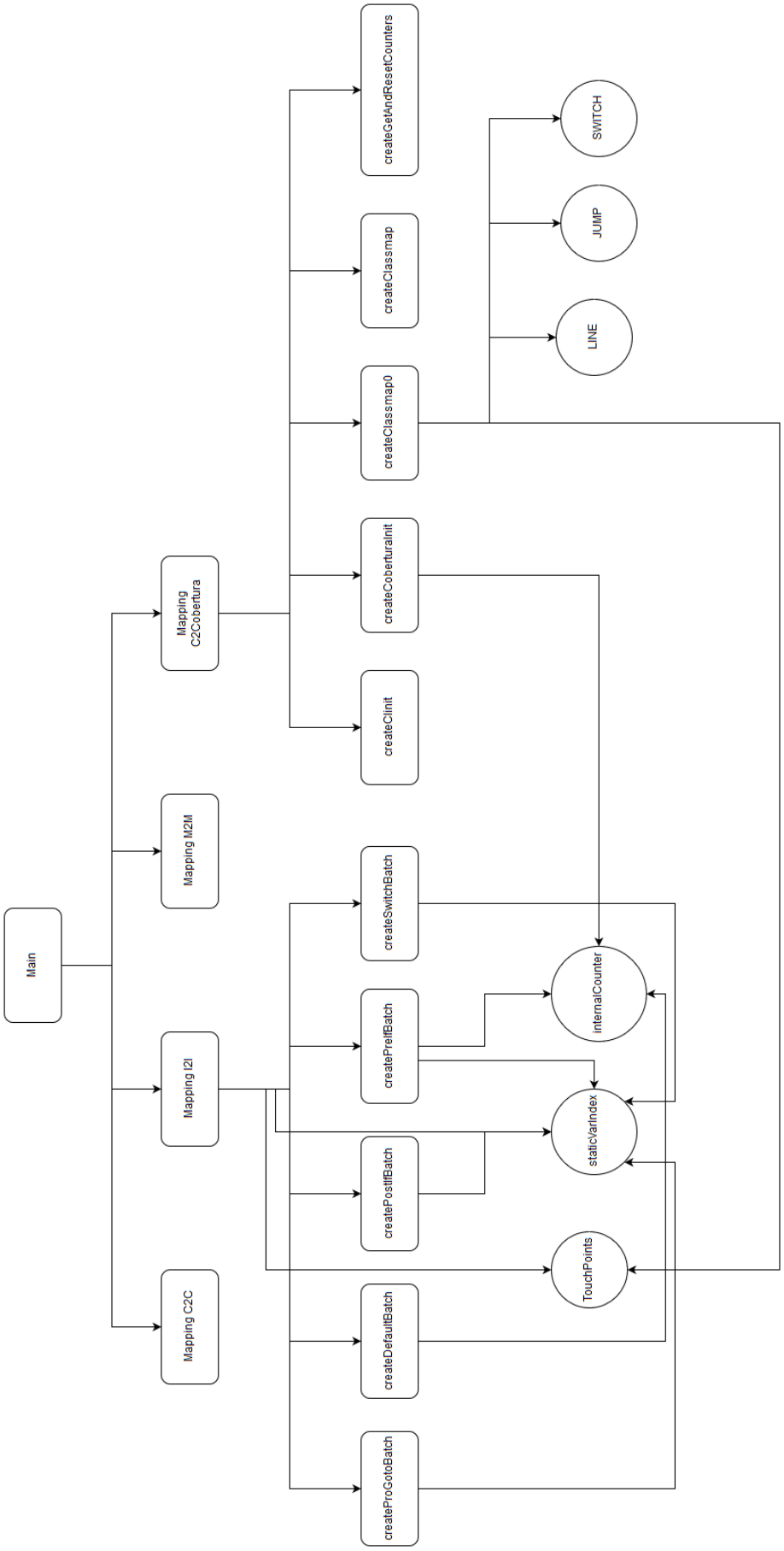


Figure 4.8: Cobertura as model transformation: LCOM4

4.6 Implementation comparison

As we now have all the necessary raw data, we can convert this using the formulae as defined in Section 3.1.1. Table 4.8 represents the transformed data of each Cobertura class, while table 4.9 represents the data of our reimplementation of the instrumentation logic.

The LoC_Q (lines of code quality) is calculated by taking the number of lines of code into account, while the F_Q (function quality) is calculated by taking the *amount* of functions in a class into account (not the content of the function itself). Using these variables and the LCOM4 as calculated in Section 4.1.1, we can calculate the resulting C_Q or class quality. For each of these values, a result closer to 1.0 means better, while a lower result means a worse class modularity quality. Numbers are rounded to the first two substantial digits.

Class	LoC_Q	F_Q	C_Q
CodeFootstamp	0.00018	0.00058	0.00038
DetectDuplicatedCodeClassVisitor	0.81	0.86	0.84
DetectDuplicatedCodeMethodVisitor	0.000059	0.0011	0.00056
DetectIgnoredCodeClassVisitor	0.96	1.0	1.0
DetectIgnoredCodeMethodVisitor	0.000084	0.0010	0.00057
BuildClassMapVisitor	0.00078	0.12	0.060
BuildClassMapTouchPointListener	0.90	0.020	0.46
AbstractCodeProvider	0.000049	0.011	0.0056
AtomicArrayCodeProvider	0.00036	1.031	0.52
FastArrayCodeProvider	0.0020	1.0	0.52
InjectCodeClassInstrumenter	0.00031	0.65	0.33
InjectCodeTouchPointListener	0.00036	0.011	0.0057
ClassMap	0.0000079	0.00041	0.00021
JumpTouchPointDescriptor	0.0062	0.65	0.33
LineTouchPointDescriptor	0.0027	1.0	0.52
SwitchTouchPointDescriptor	0.00024	0.042	0.021
TouchPointDescriptor	0.0011	0.65	0.33
FindTouchPointsMethodAdapter	0.000012	0.00070	0.00036
Average	0.15	0.40	0.27

Table 4.8: Cobertura comparable data

Class	LoC_Q	F_Q	C_Q
Full	0.0000025	0.000031	0.000017
(Alt) API	0.000041	0.00013	0.000086
(Alt) Implementation	0.0000059	0.00086	0.00043

Table 4.9: Cobertura model transformation comparable data

In summary, the instrumentation code from Cobertura has 2100 lines of code, a cyclomatic complexity of around 303 and an average class quality of 0.27. Our implementation has 409 lines of code (+190 from the API), a weighted cyclomatic complexity of 50 (+31 from the API) and a class quality of about 0.0001 (and 0.0008 for the API). Our total class quality is slightly worse than the class quality of CodeFootstamp.java from the original implementation. Our implementation has far fewer lines than the original, while also boasting a lower complexity.

In practice, the class quality comes closer to being a readability value. According to the formulae, a class would optimally have around 50 lines and 5 functions. Any number of lines deviating from that would result in lower class quality. The higher the deviation, the lower the resulting score. Our model transformation contains all the functionality in a single file, or in the best case: in two files. Since we have a higher number of lines than ideally should be in a single class, we get a lower score on our class modularity. This means that we make a trade-off. Our reimplementation is far shorter and more straightforward, yet less readable because it has all of the logic in a single file.

It is possible to split the transformation into separate parts, such as the API and the Cobertura specific parts, but even so, the class quality is a bit lackluster. There still exists room for optimisation in the model transformation code, which would reduce the number of lines. When looking at the overall data, our implementation is an improvement over the original by a large margin. The original implementation has over 2100 lines of code, with a total cyclomatic complexity of 303, spread over 18 different classes in order to work. Our new implementation might have some minor faults that can be resolved with a few lines of modifications while consisting of 599 lines of code. Our cyclomatic complexity is at 81, while almost half of that comes from constructors that can be placed in another transformation, reducing the CYC of the implementation to 50.

There are still other ways to reduce the complexity further and potentially also reduce the number of lines of code in the transformation. The use of design patterns such as the visitor pattern that ASM uses might decrease the amount of

freedom in the transformation but would move all the type-checks that have to be executed for each instruction to the back-end. Functionality such as the creation of the "__cobertura" functions could be moved to another transformation in order to further separate concerns. We can conclude that the class quality is indeed on the bad side for our reimplementation of Cobertura, but it can also be said that a model-transformation is different from a class and should, therefore, allow for a larger number of lines and functions. The difference is, however, improvable and the transformation itself performs instrumentation with far fewer lines than the original.

4.7 Conclusion

Recreating an existing implementation using another tool is hard. Even then, we managed to greatly reduce the complexity and the amount of code required to instrument a class in the same manner that Cobertura does it. The transformation still has some minor issues, but these can all be worked out with time. The origin of these issues lies in the fact that it is hard to replicate the same behaviour when encountering edge-cases. QVTo still has some minor limitations such as lack of constructor overloading, but it is very much suited for large transformations that require the creation of a large number of objects.

Mod-BEAM still has points in which it can be improved, such as error feedback that tells an approximate location of where the error occurred rather than just the error name. Another slight demerit is that instructions added from the transformation are always added to the bottom of the model rather than being positioned inline. It is possible to generate the .class file from the model and then generate the model from the .class file again to resolve this, but this is not very intuitive and only works if the transformation and generation proceed without errors. Nonetheless, usability wise it is a very usable and suitable tool for code instrumentation.

Chapter 5

Second case study: JaCoCo

This chapter contains information about the second implementation. As reasoned in Section 3.3, this implementation aims to replicate the functionality of JaCoCo, known as probes. The original implementation and our implementation is measured and then compared in order to gain an understanding of the advantages and disadvantages of our implementation. Our implementation can then be compared to the previous implementation of Cobertura which was discussed in Chapter 4. From these analysing the results we can identify the reusability of our transformation code.

5.1 JaCoCo

This section contains the analysis of the tool JaCoCo. Data about the original implementation is gathered in order to later compare it to our own implementation of the instrumentation logic.

5.1.1 Analysis

Unlike Cobertura which instruments a file in three separate passes, JaCoCo performs all instrumentation in a single pass. Functionality such as code-sanitation in order to eliminate duplicates is still required and is integrated into the visitor methods. As a consequence, JaCoCo does visit instructions multiple times, but not in entirely separate passes.

JaCoCo starts its instrumentation process by defining a ProbeStrategy. In our test-case as seen in Appendix A this would be a ClassFieldProbeArrayStrategy. There exist more strategies, such as the CondyProbeArrayStrategy, InterfaceFieldProbeArrayStrategy and LocalProbeArrayStrategy. These strategies are used to add the correct method(s) to the source class file that initializes the $jacoco probe array and for JaCoCo to be able to start the analysis process. If our code were to be

compiled using JDK 11 rather than JDK 8, it would use the CondyProbeArrayStrategy for both interfaces and classes instead. For this research, we only consider the default ClassFieldProbeArrayStrategy, as other strategies would only be viable if we wanted a 100% identical implementation while the differences it provides are extremely minimal (only one or a few generated methods depending on class type).

JaCoCo proceeds by creating a ClassInstrumenter which has the methods to track the probe count. The ClassInstrumenter then creates a DuplicateFrameEliminator which fixes ASM specific duplicate-frame issues, a ProbeInserter with the previously defined strategy and a MethodInstrumenter. The ProbeInserter generally defines what a probe looks like, how to handle the variable index and the corresponding stackmap frames. The MethodInstrumenter defines which instructions should get probes through the usage of labels, such as if, switch and multi-condition scenarios. Both the MethodInstrumenter and the ProbeInserter inherit the MethodVisitor of ASM.

JaCoCo also generates a ClassProbesAdapter which has a reference to the previously created ClassInstrumenter. This class calls the VisitMethod function that was defined in the ClassInstrumenter so that for each method duplicate frames are eliminated, probes are defined and the probes are inserted at the correct position.

The ClasssProbesAdapter proceeds by running a MethodSanitizer for every method end with a probe strategy of type ”EMPTY_METHOD_PROBES_VISITOR”. This method sanitiser fixes, as per the official documentation, two potential issues with the Java bytecode. The first issue is to remove JSR/RET instructions, which are deprecated since Java 6, without breaking the class file. The second issue is to fix data generated by other tools that contain invalid offsets or other attributes. Since these issues are not directly related to our implementation, they will not be incorporated into the calculation of NCLoC, CYC and other statistics. The MethodSanitizer also calls LabelFlowAnalyzer, which is used in order to determine if labels are connected to each other or require a probe point.

Table 5.1 shows the analysis data for the ”flow” package, which contains classes such as LabelInfo and the MethodProbesAdapter. Table 5.2 shows the analysis data for the ”instr” package, which contains classes such as the ProbeInserter or various Probe strategies. Only classes that are relevant to instrumenting a single normal class file are considered for analysis.

Class	LoC	Function count	Average CYC	Total CYC	LCOM4
ClassProbesAdapter	66	6	1.33	8	1
ClassProbesVisitor	15	4	1	2	1
FrameSnapshot	41	4	2	8	1
LabelFlowAnalyzer	151	21	1.48	31	1
LabelInfo	104	20	1.65	33	1
MethodProbesAdapter	134	12	2.25	27	1
MethodProbesVisitor	35	8	1	8	1
MethodSanitizer	27	3	1.67	5	1

Table 5.1: JaCoCo flow

Class	LoC	Function count	Average CYC	Total CYC	LCOM4
ClassFieldProbeArrayStrategy	68	6	1.17	7	1
ClassInstrumenter	47	5	1.2	6	1
DuplicateFrameEliminator	89	15	1.07	16	1
MethodInstrumenter	137	11	3	33	1
ProbeCounter	31	5	1.2	6	1
ProbeInserter	90	9	1.89	17	1

Table 5.2: JaCoCo instr

5.1.2 Instrumentation

The actual probe points that JaCoCo instruments are significantly more straightforward than those of Cobertura. While Cobertura implements multiple batch types, such as a counter ID in a variable, creating a local copy, or the functionality to set this local copy to zero, JaCoCo only implements a single batch type. The only thing that JaCoCo does is to regularly add in a probe point which sets an index of a boolean

array to true. This probe point is added under certain conditions, which are stated below:

- Each line gets a probe at the end of the line (only added when the label (or next line) is a method invocation)
- Each return statement gets a probe at the start of the line
- When the next instruction is a GOTO instruction, the new batch should be prepended to the GOTO.
- An if instruction gets a default batch on the ”false” branch.

The results of this can be seen in Figure 5.1.

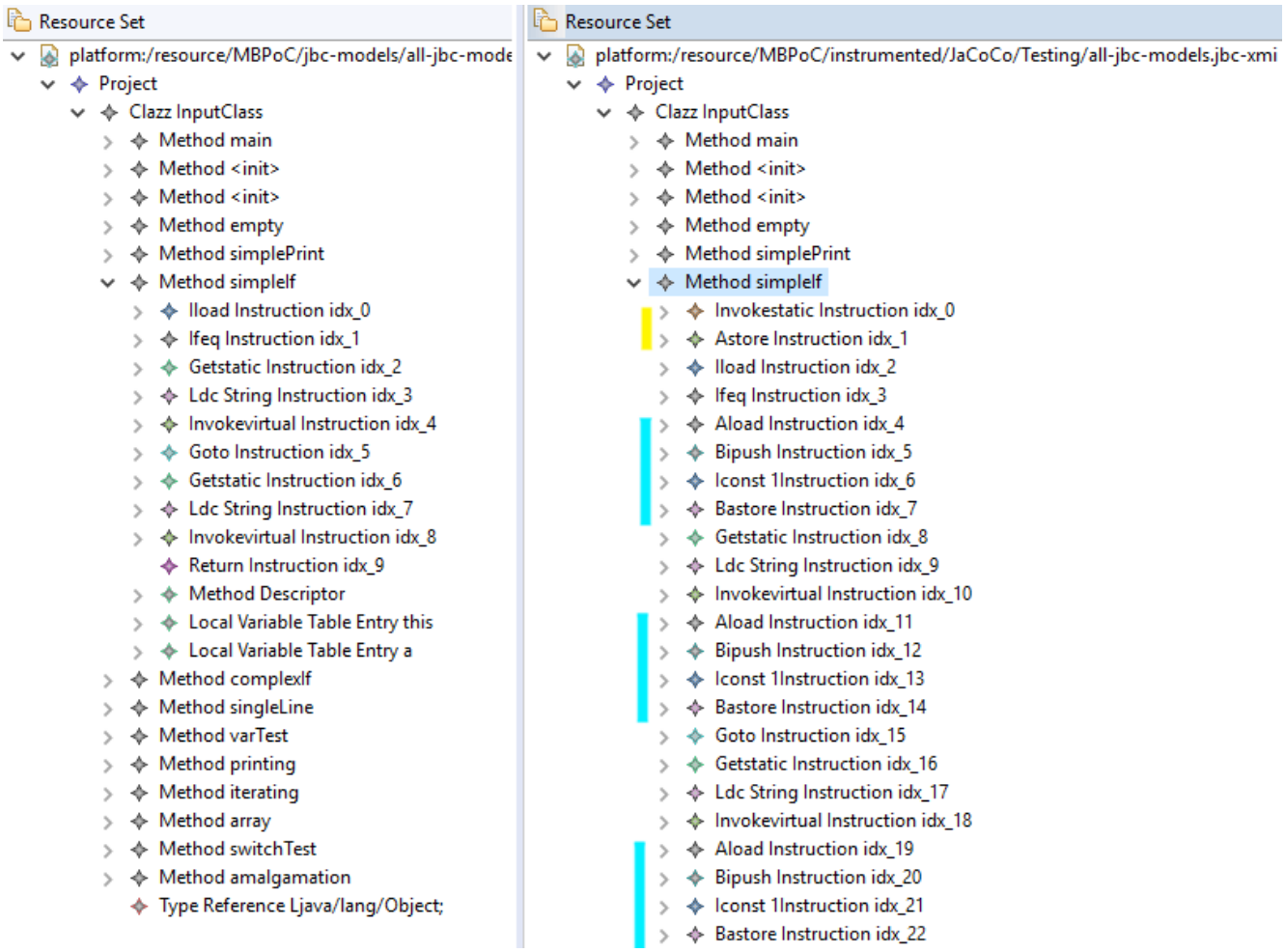


Figure 5.1: Instrumentation analysis

Multiple if statements on a single line are also an exception to the aforementioned rule When a multi-condition occurs (such as ”a and b or c”). This is done by checking if multiple control paths lead to a single label. In the event that this occurs, all related

if statements are changed into their inverted equivalent. This means that instructions such as IFGE are changed into IFLT and IFEQ changes into IFNE. For the false path, an instruction is inserted, together with a GOTO instruction. The GOTO instruction has an unconditional edge to the original true end, and a resume edge for the original false end. The true path gets a reference to the original false end. The bytecode actions can be seen in Figure 5.2. Listing 5.1 shows the original decompiled class, while Listing 5.2 shows the instrumented (by JaCoCo) decompiled class.

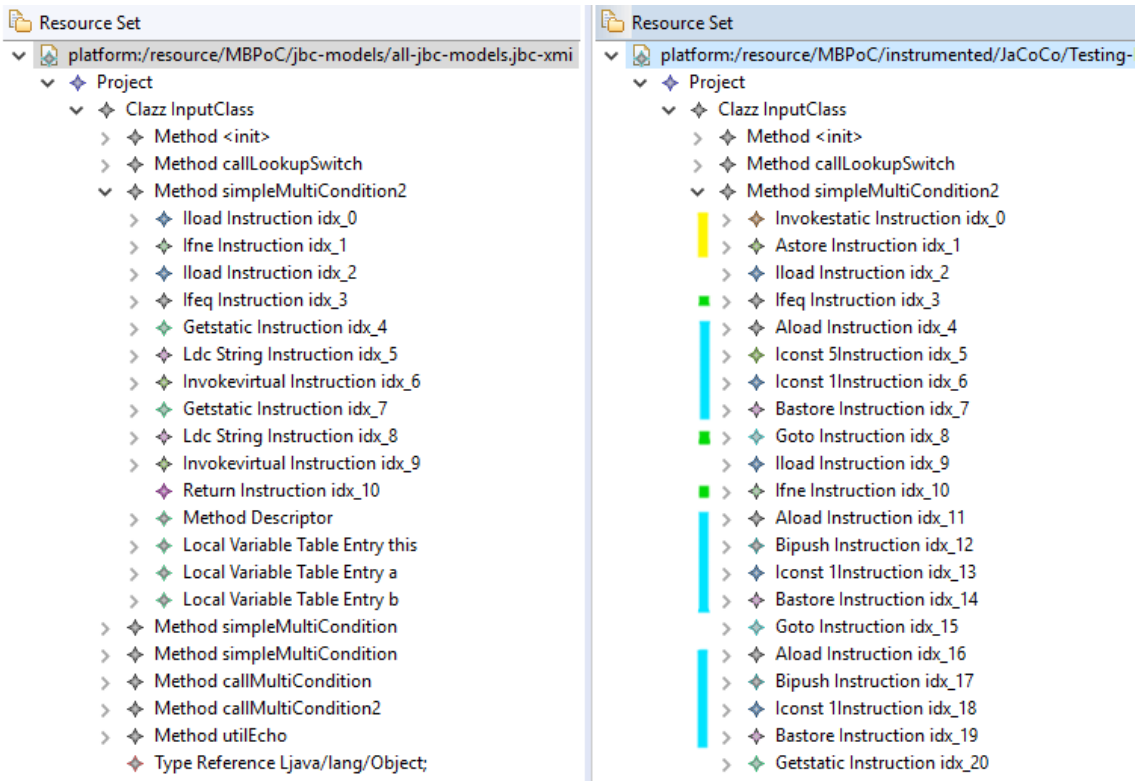


Figure 5.2: Instrumentation multi-condition

```
public void simpleMulti(boolean a, boolean b) {
    if (a || b) {
        System.out.println("true");
    }
    System.out.println("false");
}
```

Listing 5.1: multi-condition input

```
public void simpleMulti(boolean var1_1, boolean var2_2) {
    block2 : {
        var3_3 = InputClass.$jacocoInit();
        if (a == false) break block2;
        var3_3[5] = true;
        ** GOTO lbl10
    }
    if (b == false) {
        var3_3[6] = true;
```

```
    } else {
        var3_3[7] = true;
        lbl10: // 2 sources:
        System.out.println("true");
        var3_3[8] = true;
    }
    System.out.println("false");
    var3_3[9] = true;
}
```

Listing 5.2: multi-condition results

5.2 JaCoCo as model Transformation

This subsection contains information about the reimplementation of JaCoCo's instrumentation functionality as model-transformation using Mod-BEAM. Control-flow handling and code quality are similar to the implementation of Cobertura as model transformation, as described in Section 4.2.

5.2.1 ID Ordering

The ID sequencing for this variable is done by traversing the code through the order of instructions in a class rather than by following the control flow of the code, which results in an ID order which is identical to our own. This means that as long as we correctly implement the same rules that JaCoCo follows, the result of our implementation will be identical to that of JaCoCo.

5.2.2 Instruction rules

JaCoCo can be implemented in a single pass. We simply need to follow the rules as defined in Section 5.1.2. A batch function is created in order to define the setting of a value at an index of a boolean array. All we need to do is handle the correct situations at which a probe needs to be added. As we are working with a model transformation, we no longer have access to a label. Instead, we have access to information that a label would normally such as an offset, in the form of an outEdge and a next instruction. As such, we can define a valid probe point by checking if the next instruction is not a variable instruction such as Iinc, Iload and Bipush among others. We can follow this up by creating some rules of our own:

- If the current instruction is an if instruction of any type, add a batch
- If the next instruction is on a different line, check if it is a valid probe point or a GOTO instruction. If so, add a batch.

- If the next instruction is a return instruction, add a batch as prepend.

For the normal new instructions, we can simply add a batch at the end. We do have to check, however, if the instruction has multiple inEdges that lead towards it. When the next instruction has multiple incoming edges, it means that the instruction is the target of a jump. For our implementation, this means that when this occurs we do not want to link every incoming instruction to the first instruction of our batch. Instead, the desired result is to link only the resume edge to the new batch, while keeping the jump to the original target. This makes it so that the new batch remains scoped within the if or else block, while the next target remains outside of this scope.

These rules do not yet handle multi-condition conditions. Those cases, together with try/catch blocks were not fully supported in our implementation of Cobertura. As such, they will also not be supported in our implementation of JaCoCo. We have however created a pseudo-implementation to prove that it is very much possible to implement this and in order to hold into account how many lines of code and how complex it would be to actually implement this. JaCoCo uses their LabelFlowAnalyzer in order to determine if multiple labels are on the same line and a multi-condition has been found. The way that this is determined is a bit unclear, but we can create our own way of handling it. By simply adding all if-statements to a list we can determine if more than one if-clause is on a single line. If this is the case, we can perform the inversion strategy exactly the same as JaCoCo does this.

If a new line is traversed, we can simply clear out the list. An issue with this is that JaCoCo does not seem to invert every if instruction that shares the same line as another if instruction. Research proved that every if instruction without a corresponding GOTO for the 'true' branch (e.g. the jump) was inverted. With the pseudo-implementation, we do have to be careful with our ID generation as the IDs will have to be generated in sequence before the code for the next line is executed. The latest version of our implementation contains the logic in order to invert if-statements similar to how JaCoCo does this but misses some batches to create at the appropriate positions. An example of if statement inversion is visible in Figure 5.2.

The implementation itself for if-inversion is quite straightforward. For each possible type of if-statement, an instruction of the opposite type is created. The in- and out-edges are copied onto the new instruction, and any incoming edge is redirected to the newly created if-statement. The actual complexity lies in the logic where the true and false edges of the newly created if-statement are inverted. This is because not only are the edges inverted and batches added, the new true edge (originally the false edge) also inserts a GOTO instruction with appropriate edges in order to check which of the if-instructions have actually been traversed.

5.3 Assessing functional equality

Assessing functional equivalence is yet again a two-step process. We first prove that the variables that we added have no impact on the original code, as we have a minor difference with the way we set variables. The second step is to prove that the output of a JaCoCo instrumentation task is identical to the output of our instrumentation as model transformation.

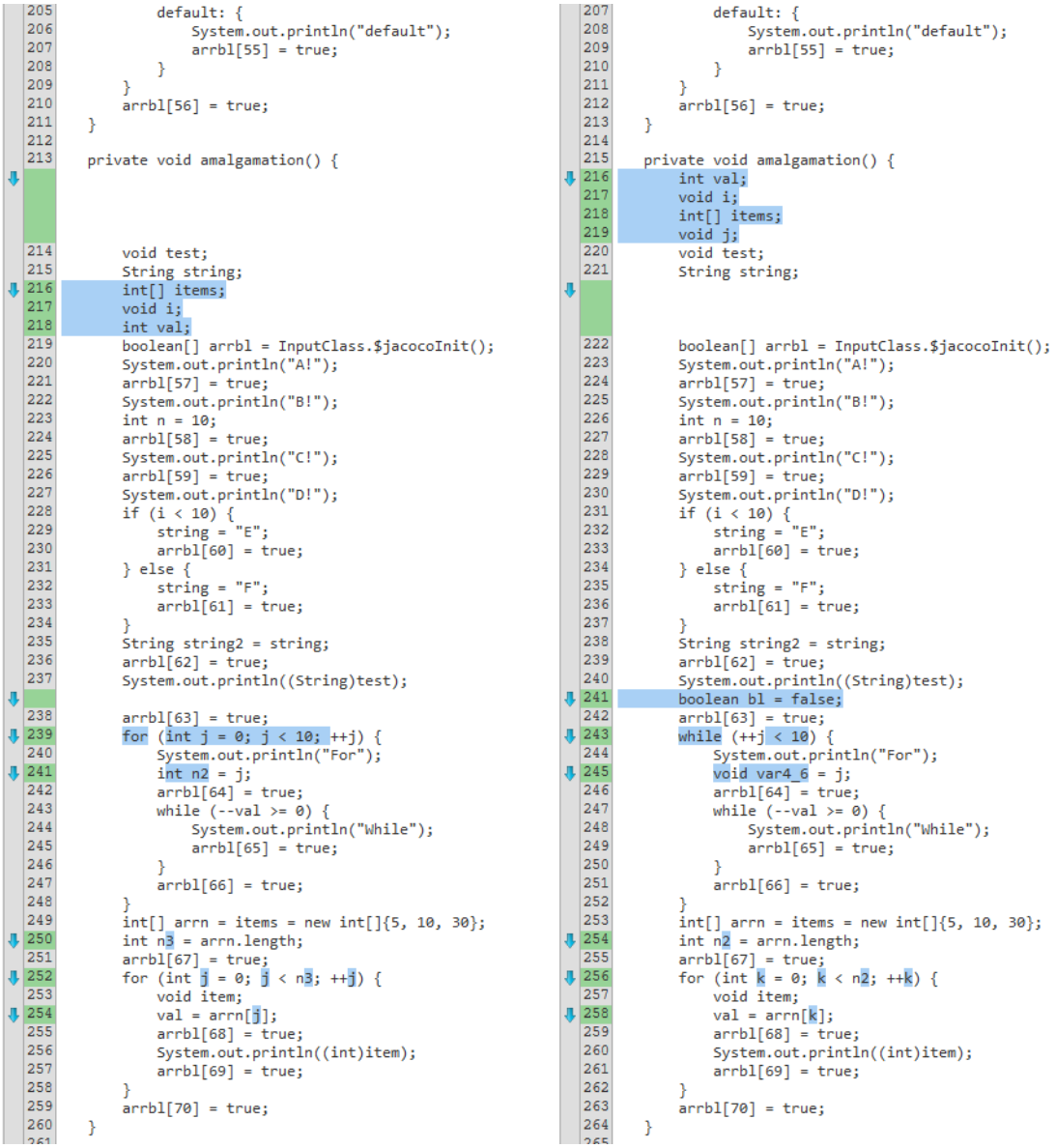
Assessing that our variables are not in the way of the existing code cannot be done without making some minor modifications. This is because the initialization method for the Boolean array creates a reference to a JaCoCo function that we cannot access. While we can work our way around this by modifying how we generate the $jacocoInit function, it is far easier to simply create the necessary function in plain Java. We then add an if clause around our transformation so that instructions belonging to this method are not modified. This will allow us to create the appropriate initialization function for the ID tracking (in the source code), without relying on any JaCoCo specific functionality. Listing 5.3 shows the alternative Java function that provides the necessary data. Similar to Cobertura in Section 4.3, the results are identical to the unmodified output of our test class (Appendix A).

```
private static transient /* synthetic */ boolean[] $jacocoData;

private static /* synthetic */ boolean[] $jacocoInit() {
    boolean[] arrbl = $jacocoData;
    if (arrbl == null) {
        arrbl = $jacocoData = new boolean[71];
    }
    return arrbl;
}
```

Listing 5.3: Alternative init

The second part of our verification process is more straightforward. We simply run our generated .class file through a decompiler and compare the result to the .class file that was generated by JaCoCo itself. If the results match for our decently-sized test-case, then we can assume that the implementation is sufficiently equivalent. The results of this can be seen in Figure 5.3, which shows that the only differences lie in the manner that variables are handled. The difference in how variables are handled (a matter of simply using a high index without modifying the already existing indices on the bytecode level), leads to the differences that can be found in the figure that was mentioned above. Our ID sequence and instruction positions are completely identical.



```
205             default: {
206                 System.out.println("default");
207                 arrbl[55] = true;
208             }
209         }
210         arrbl[56] = true;
211     }
212
213     private void amalgamation() {
214         void test;
215         String string;
216         int[] items;
217         void i;
218         int val;
219         boolean[] arrbl = InputClass.$jacocoInit();
220         System.out.println("A!");
221         arrbl[57] = true;
222         System.out.println("B!");
223         int n = 10;
224         arrbl[58] = true;
225         System.out.println("C!");
226         arrbl[59] = true;
227         System.out.println("D!");
228         if (i < 10) {
229             string = "E";
230             arrbl[60] = true;
231         } else {
232             string = "F";
233             arrbl[61] = true;
234         }
235         String string2 = string;
236         arrbl[62] = true;
237         System.out.println((String)test);
238         arrbl[63] = true;
239         for (int j = 0; j < 10; ++j) {
240             System.out.println("For");
241             int n2 = j;
242             arrbl[64] = true;
243             while (--val >= 0) {
244                 System.out.println("While");
245                 arrbl[65] = true;
246             }
247             arrbl[66] = true;
248         }
249         int[] arrn = items = new int[]{5, 10, 30};
250         int n3 = arrn.length;
251         arrbl[67] = true;
252         for (int j = 0; j < n3; ++j) {
253             void item;
254             val = arrn[j];
255             arrbl[68] = true;
256             System.out.println((int)item);
257             arrbl[69] = true;
258         }
259         arrbl[70] = true;
260     }
```

```
207             default: {
208                 System.out.println("default");
209                 arrbl[55] = true;
210             }
211         }
212         arrbl[56] = true;
213     }
214
215     private void amalgamation() {
216         int val;
217         void i;
218         int[] items;
219         void j;
220         void test;
221         String string;
222         boolean[] arrbl = InputClass.$jacocoInit();
223         System.out.println("A!");
224         arrbl[57] = true;
225         System.out.println("B!");
226         int n = 10;
227         arrbl[58] = true;
228         System.out.println("C!");
229         arrbl[59] = true;
230         System.out.println("D!");
231         if (i < 10) {
232             string = "E";
233             arrbl[60] = true;
234         } else {
235             string = "F";
236             arrbl[61] = true;
237         }
238         String string2 = string;
239         arrbl[62] = true;
240         System.out.println((String)test);
241         boolean bl = false;
242         arrbl[63] = true;
243         while (++j < 10) {
244             System.out.println("For");
245             void var4_6 = j;
246             arrbl[64] = true;
247             while (--val >= 0) {
248                 System.out.println("While");
249                 arrbl[65] = true;
250             }
251             arrbl[66] = true;
252         }
253         int[] arrn = items = new int[]{5, 10, 30};
254         int n2 = arrn.length;
255         arrbl[67] = true;
256         for (int k = 0; k < n2; ++k) {
257             void item;
258             val = arrn[k];
259             arrbl[68] = true;
260             System.out.println((int)item);
261             arrbl[69] = true;
262         }
263         arrbl[70] = true;
264     }
```

Figure 5.3: Instruction output comparison

That the instructions are identical in our test case does not prove that the implementation covers all types of instructions or scenarios. As mentioned in Section 5.2, we do not handle multi-condition scenarios. Similar to Cobertura, we also do not handle try/catch blocks. There might be other situations that are not covered by the test-case. As our reimplementation significantly differs from the original JaCoCo code, those differences can only be uncovered by using the implementation in practice. As such, for an implementation to prove that the tool is feasible this implementation should very well suffice.

5.4 Reimplementation analysis

In order to compare our implementation to the original implementation, we have to gather the necessary data, such as the NCLoC, CYC and LCOM4 values. The constructors and functions that are necessary for handling control flow actions have been moved to a separate transformation called the ”APITransformation”, while the implementation-specific code is now stored in the ”JacocoTransformation” which extends the APITransformation.

By trimming all empty lines, we now have an API that consists of 306 NCLoC. This is more than the original 190 as it was in the Cobertura implementation as measured in Section 4.5 because we added additional constructors for types that were missing but were now required. The complete list of methods in the API and their complexity can be seen in table 5.3. The reimplementation itself has a NCLoC of 128. The complexity of the implementation can be found in table 5.4.

Class	**Total CYC**
Constructors (50)	50
API Methods (4)	9
Total	59

Table 5.3: Reusable QVTo API

Class	**Total CYC**
Main	
Constructor BatchContainer	1
Helper linkWithinJumpBlock	4
Mapping C2C	1
Mapping M2M	1
Mapping I2I	11
Query isProbePoint	1
Helper createDefaultBatch	1
Mapping C2JaCoCo	1
Total	21

Table 5.4: JaCoCo as model transformation

The LCOM4 is as usual 1.0, as there are only two variables. The first variable is the staticVarIndex at which integers are placed, while the second variable is the internalCounter which tracks ID generation.

5.5 Implementation comparison

As we now have all the necessary raw data, we can convert this using the formulae as defined in Section 3.1.1. Table 5.5 represents the transformed data of each JaCoCo class, while table 5.6 represents the data of our reimplementation of the instrumentation logic.

Similar to the comparison of our reimplementation with Cobertura in Section 4.6, we can now calculate the necessary data in order to measure the class quality. A higher C_Q or ”class quality” means higher class modularity. In practice, it comes closer to being an indicator of how readable a class is. The corresponding formulae can be found in Section 3.1.1.

Class	LoC_Q	F_Q	C_Q
ClassProbesAdapter	0.0034	0.65	0.33
ClassProbesVisitor	0.56	0.86	0.71
FrameSnapshot	0.89	0.86	0.87
LabelFlowAnalyzer	0.000079	0.00049	0.00028
LabelInfo	0.00029	0.00058	0.00043
MethodProbesAdapter	0.00012	0.0045	0.0023
MethodProbesVisitor	0.81	0.042	0.43
MethodSanitizer	0.71	0.69	0.70
ClassFieldProbeArrayStrategy	0.0027	0.65	0.33
ClassInstrumenter	0.96	1.0	1.0
DuplicateFrameEliminator	0.00056	0.0017	0.0011
MethodInstrumenter	0.00011	0.0068	0.0035
ProbeCounter	0.76	1.0	0.90
ProbeInserter	0.00053	0.02	0.010
Average	0.34	0.42	0.38

Table 5.5: JaCoCo comparable data

Class	LoC_Q	F_Q	C_Q
API	0.000012	0.000023	0.000018
Implementation	0.00013	0.020	0.010

Table 5.6: JaCoCo model transformation comparable data

In summary, the instrumentation code from JaCoCo has 1035 lines of code, a cyclomatic complexity of around 207 and an average class quality of 0.4. Our implementation has 128 lines of code (+306 from the API), a weighted cyclomatic complexity of 21 (+59 from the API) and class quality of about 0.01 (and far lower for the API). Our implementation is about as large as one of the larger classes of the original implementation. This means that our implementation is far more simple and far more straightforward. We lose some readability as the class quality is not that high, but that is to be expected when nearly all of our code is in a single file when compared to class modularity in object-oriented languages.

We still have some deviations in the result, but these are all non-critical for our proof. Even though it is not complete, even the hypothetical implementation of multi-condition conditions would add a single function to inverse if-clauses (38 NCLoC), a detection mechanic (circa 2 to 5 NCLoC) and a bridge mechanic to make it work (circa 20 NCLoC). Other issues can be fixed more easily, requiring a couple of code lines at most.

As explained in Section 4.6, the class quality is based on the assumption that an optimal amount of lines of code is around 50 and an optimal amount of functions in a single class is 5. Similar to the comparison of Cobertura, we trade in readability for the sake of a far smaller implementation. Unlike Cobertura, however, our reimplementation of JaCoCo is not even that much larger or unreadable than some of the classes in the original implementation. Classes such as the LabelFlowAnalyzer have a worse LoC quality and a worse function quality than our entire implementation. The counterargument for this is that our implementation does not mimic the entirety of what the original JaCoCo implementation supports, such as try/catch conditions, multi-condition statements or other undefined scenarios. In the code of JaCoCo, this functionality is extremely simple, containing only about 30 lines for the inversion and about 13 lines to perform the insertion of the correct instructions. This is however encapsulated by a more complex label tracking system in order to detect if multiple control paths lead to the same label (part of the LabelInfo class).

Our CYC is also lower than the original. While we have a total complexity of 21 for our entire implementation, JaCoCo boasts a complexity of 207 spread across all relevant classes. The calculated complexity is biased slightly in our favor due to the fact that only the if / else statements are counted (Weighted Cyclomatic Complexity) rather than the McCabe cyclomatic complexity. The favor originates from the fact that we have some complex conditional statements which we use to determine if an instruction is worthy of a probe or not. Regardless, the cyclomatic complexity would still be far less than that of the original implementation. If the API itself were factored in then the total complexity would come much closer to that of the original, being a CYC of 80 due to the sheer constructor count required to properly instantiate

objects. In conclusion, our implementation does nearly everything JaCoCo does while remaining the same size as one of JaCoCo's larger classes, rather than the entire instrumentation package.

5.6 Reimplementation comparison: Cobertura and JaCoCo as model transformations

In this section, the reimplementations of Cobertura and JaCoCo are compared in order to check how much code is reusable. Besides this, the performance of the newly created reimplementations is measured against the performance of the original implementations of Cobertura and JaCoCo.

5.6.1 Reusability

The original implementation of Cobertura and JaCoCo only really share the same concept of instrumenting code in order to make the code measurable. The actual implementation differs so much that it would be very hard to find any reusability. Our implementation of the instrumentation functionality of Cobertura provided some base constructors and methods to facilitate code reuse and reduce the amount of bloated code from setting individual variables on a separate line. Our implementation of JaCoCo expanded upon this base, which caused the creation of even more constructors and helper methods. This API was then separated into a different file which could be extended by either the Cobertura or the JaCoCo implementation. Even then, we still have some visible reuse between the implementations. Both implementations require the use of a BatchContainer which stores the first and last instructions of an instruction batch in order to keep the control-flow correct. Both implementations also have an internal counter in order to generate unique IDs and a set of if-cases in order to add instruction batches to the corresponding instructions at the correct place. While the actual batches and positions of the batches are different (before instead of after a line, different instructions in a batch, various batch types), the conditions that were monitored were often the same (if/switch/line number). This means that overlap in our API and implementation could technically be further separated into the API, overlapping implementation functions and the implementation itself. While this might be possible to do, it would be more interesting to add functionality to the API that mimics the behaviour of Cobertura, such as onSwitch, onIf, onGoto and the likes. These cases could then potentially be overridden by the inheriting transformation, which could cause the transformation to be even more complete.

5.6.2 Performance

Performance time is measured by taking the execution time of the instrumentation process into account. This means that any overhead that stems from calling the execution is ignored (e.g. when running the instrumentation from a test framework or ant-script). In order to do this, a few simple Java operations are added to the source-code of Cobertura and JaCoCo. The code used to measure is as simple as taking the current time in milliseconds and checking the difference after the instrumentation process is finished. In order to do the same in QVTo, a Java black-box is created, which is essentially a self-made library that allows for Java calls from the QVTo environment.

To reduce potential variables, any remaining logging is removed from the original code of Cobertura and JaCoCo. The tests are run on the same disk, with the system under approximately the same amount of load. Tests are executed on ”InputClass.java” as found in Appendix A, Listing A.1. Each test is run around ten times in order to determine a minimum, maximum and average. The results can be found in Table 5.7.

Source	**Minimum**	**Maximum**	**Average**
Cobertura	161	166	163
Cobertura Reimplementation	55	88	73
JaCoCo	74	81	76
JaCoCo Reimplementation	25	36	31

Table 5.7: Performance in milliseconds

For this small test-case, our reimplementation is faster than the original implementations. This is however only the case because the original class is already transformed into the corresponding model, and because we do not take into account the time that might be required to transform the model back to a valid Java .class. Besides this, ant generated a large amount of overhead (around 150ms), which makes deciding how the transformation should be called important in terms of performance. Alternatively, the reimplementations could be called from Java, but this requires some additional configurations that are poorly documented.

5.7 Conclusion

From the analysis, we can conclude that our reimplementation contains less code and is less complex than the original implementation by JaCoCo. While the class quality was poor in our reimplementation of Cobertura due to the number of lines in a single file, our class quality for JaCoCo was decent. Our reimplementation of JaCoCo still misses some functionality with regards to inverting if statements at the correct position, but this was taken into account when calculating the results for the analysis.

As mentioned in Section 3.1, we want to infer how easy it is to create a new concept from scratch and how hard it is to add something to an existing project based on the gathered metrics. We can determine that creating a new product from scratch would be less complicated in terms of CYC and would take up less size (NCLoC). The conclusion for this is that making a new product would be easier to do by using Mod-BEAM. Extendibility is different, as our implementation in Cobertura had a far lower class quality than average, while for the implementation of JaCoCo it scored better than the worst class. Extendibility would, therefore, be a bit dependent. Through the use of some patterns as mentioned before, the rules to iterate through the code could be simplified to properly define when and where something is added (e.g. ”beforeVar”, ”afterJump”). This would improve extendibility by simplifying the rules and moving some of the lines and complexity further to the back-end of the transformation.

Chapter 6

Hypothetical case study: AspectJ

AspectJ is a tool that differs from the previously implemented tools such as Cobertura (Chapter 4) and JaCoCo (Chapter 5). AspectJ is a tool for AOP rather than CCA. The goal of AspectJ is to allow for modularisation through the use of a Java overlay, namely dynamic join points. AspectJ allows users to define functionality and where this functionality should be ”weaved” into their code. This weaving process makes use of bytecode instrumentation through a modified version of BCEL.

6.1 Analysis of AspectJ

AspectJ defines several important concepts that have to do with bytecode instrumentation. These concepts are pointcuts (selective join points and values at those points), advice (additional actions to take at join points in a pointcut), inter-type declarations (”open classes”) and aspects (”a modular unit of crosscutting behavior, comprised of advice, inter-type, pointcut field, constructor and method declarations” [9]). In simplified terms, this means that a Join Point is a point in the program flow, for example, a call to a method (dynamic) or a method or class definition (static). A pointcut is a predicate that matches join points. An ”advice” is quite simply the new code to be executed, e.g. before or after each predicate. An Aspect is then an object which contains the defined pointcut, join point and advice functionality. An example from the original tutorial can be found in Listing 6.1.

```
public aspect LoggingAspect {
    pointcut balanceAltered(int i) :
        call(public void Account.deposit(int)) ||
            call(public void Account.withdraw(int));

    before(int i) : balanceAltered(i) {
        System.out.println("The␣balance␣changed");
    }
}
```

Listing 6.1: AspectJ Tutorial

Consider a banking application, which consists of the original source code and the AspectJ functionality as depicted in Listing 6.1. When the code is compiled without using AspectJ, the resulting code simply performs all account modifications without generating any output. If the AspectJ code is compiled and therefore ”weaved” at the defined join points. The resulting .class file has then been modified such that each call to deposit(int) or withdraw(int) first make a call to balanceAltered(i). This means that each withdraw or deposit action will create an output on execution, while the original source code of the banking application

Internally, join points in AspectJ contain respective ”shadows” in the original source code of the application. These shadows may be of different types, where each shadow represents a location where the new code should be inserted. In the weaving process, each identified shadow is of the type ”BcelShadow”, which can be found in the ”weaving” package. Each shadow has a ”Kind”, which may be a method, constructor or nearly any other Java type. The weaving process usually starts in the ”WeavingURLClassLoader”. The class loader proceeds by making a ”WeavingAdaptor”. This class then creates a ”BcelWorld” object and a ”BcelWeaver” with the ”BcelWorld” as reference. This ”BcelWeaver” contains the functionality to find pointcuts, optimize them, detect duplicity and to actually weave them together.

6.2 Discussion: reusability

While in Part I the conclusion was reached that there might be potential reusability between the implementation of a CCA tool and an AOP tool due to the concept of ”touch-points”, the implementations of this concept in Cobertura and JaCoCo have proven that there exists a fundamental difference. The concept of ”touch-points” in CCA can be considered as key-points which should be considered as important points that define the actions or branching in the control flow for the execution of a program. In AspectJ, a ”join point” is a part of a predicate that defines for which piece or pieces of code a batch of ”advice” code should be added. Due to the difference in nature for these points and based on the implementation of the instrumentation functionality of Cobertura and JaCoCo, the conclusion can be reached that there would be little overlap in the actual code between these tools, whether in the original or in an implementation through Mod-BEAM. An implementation as model transformation through Mod-BEAM would share the same functionality for constructors and control-flow handling as defined in the earlier implementations.

6.3 Discussion: weaving implementation

Making use of a model-transformation language in combination with Mod-BEAM allows for low-level, while also very straightforward access to bytecode. As such, it would be very much feasible to implement the weaving process similar to AspectJ. There would be differences in the execution process, as the model of the code would have to be generated before the transformation can be executed, but creating functionality that takes a set of defined points and adds it into the corresponding methods is possible. A possible implementation would consist of the following steps:

- For each method that has an Aspect annotation (mapping)
  - Filter out pointcuts
  - Add pointcuts to a list of tracked targets
  - Filter out advice
  - Store advice with the tracked targets
- For each method (mapping)
  - Check if method name matches any of the tracked targets (static join point)
  - Add the advice before the first instruction / before the last return instruction depending on the type
- For each invoke instruction (mapping)
  - Check if method invoke matches any of the tracked targets (dynamic join point)
  - Add the advice before / after depending on the type

An implementation as such could be a simplified implementation of how instructions can be weaved at the correct position. The method to determine where the advice should be weaved could depend on the type of join point that we are looking for. Static join points would include method definitions, while dynamic join points would be method invocations. Implementing the base functionality of weaving would not mean that the implementation does everything that AspectJ does, just the main part of the weaving process for Java code. The performance of the actual weaving process might differ, but the instrumented code should be identical to the code generated by AspectJ itself.

6.4 Potential issues

There are still some issues that were encountered during the implementation of instrumentation of CCA tools that might occur here. One such issue might be that if the content of a function has to be merged with an existing function that a hard-coded variable index might not suffice. If this is the case then the variable index would have to be computed or inserted by moving other variable indices out of the way. Fortunately, AspectJ only in-lined code in early versions, and currently weaves advice as separate function calls [10]. Other functionality to watch out for would be to compare how AspectJ handles duplicate names and inheritance.

Another issue might be that since Mod-BEAM is an early tool, not all code situations have exhaustively been tested, which means that errors could occur. This would mean that the stability of the tool in a broad scope of code scenarios can still be further tested.

6.5 Conclusion

While only the important parts of a possible implementation have been described, these parts are realistically implementable. There are some issues to watch out for during the implementation, but none of the mentioned issues should be severe enough to prevent a functional product. Stability of Mod-BEAM might be a concern that could still be further tested, but it works in most general use-cases. AspectJ is a large product while relying on a modified version of a dated product (BCEL), which makes it hard to fully understand. The results are however very much reproducible using Mod-BEAM.

Chapter 7

Conclusion

This chapter consists of two sections, namely the conclusions to the various research questions, a discussion on the results and recommendations. The conclusion walks through each Research Question from Section 2.1.2

7.1 Conclusions

We answered the following research questions:

- *In which aspects can Mod-BEAM be analysed in order to compare it to other approaches?*

We answered this question by defining a set of criteria based on existing research. In Section 3.1 we defined that usability can be measured by taking the lines of code, complexity, modularity, ease to create (functionality from scratch) and extendibility into account. We also defined that reusability can be measured by taking the amount of reusable code across reimplementations into account.

- *What are the advantages and disadvantages of declarative or imperative transformations through Mod-BEAM and which languages are viable?*

This question is answered in Section 3.2. We conclude that declarative languages hold the advantage in performance and that they hold the advantage on how easy it is for computers to analyse the code for completeness. Creating new objects is easiest when making use of an imperative language. A hybrid language holds most of the important advantages of both imperative as well as declarative languages for our goal, as a hybrid language is fast as well as suitable for object creation.

We concluded that QVTo would be a suitable starting point and that ETL could be a suitable language to migrate towards once we established some base functionality. In Section 4.4 we conclude that ETL lacks the necessary functionality in

order to maintain readable code (constructors) and the necessary developer environment support to highlight incorrect code or appropriate suggestions. From this, we reasoned that QVTo is, as compared to other languages, the most suitable model-transformation language. While imperative model to model transformations might not be optimal in terms of performance, it wins out in terms of usability or ease-of-use due to the completeness of the language implementation. There are still some improvable points, such as constructor overloading which would improve overall readability, but it is sufficient for common usage.

- *Which concepts of which tools have potential reusability that cannot be utilised in the current implementation?*

We analyse the potential reusability by using the results of Part I as a base. This is done in Section 3.3. We conclude that Cobertura and JaCoCo (CCA tools) both reimplement the gathering and instrumenting of touch-points and therefore could potentially share code. We also reason that AspectJ (an AOP tool) could be a suitable extension of this functionality as it makes use of hook- or entry-points.

- *How complex is a model-driven solution of instrumentation-based functionality for the viable tools and concepts?*

After reimplementing the instrumentation functionality of both CCA tools Cobertura and JaCoCo using QVTo and Mod-BEAM, we analysed the results. The results of our reimplementation of the instrumentation functionality of Cobertura can be found in Section 4.5, and the results of our reimplementation of the instrumentation functionality of JaCoCo can be found in Section 5.4.

For our reimplementation of Cobertura, our implementation has 409 lines of code (+190 from the API), a weighted cyclomatic complexity of 50 (+31 from the API) and a class quality of about 0.0001. The low class quality can be attributed to the model transformation having all logic in a single file, while it is distributed in the original implementation. There are still improvements that can be made to the class quality, such as the use of patterns or improvements in the transformation to reduce lines of code.

For our reimplementation of JaCoCo, our implementation has 128 lines of code (+306 from the API), a weighted cyclomatic complexity of 21 (+59 from the API) and class quality of about 0.01 (and far lower for the API). The class quality is higher than in our reimplementation of Cobertura, as the model transformation for our reimplementation of JaCoCo is far smaller.

- *How does the model-driven implementation compare to the original implementation?*

We compare our reimplementations to their original counterparts in Section 4.6 (Cobertura) and Section 5.5 (JaCoCo). We came to the conclusion that our reimplementations of Cobertura and JaCoCo are significantly smaller while also being far less complex compared to the original implementations. While the reimplementations still missed some functionality, we made appropriate estimates of the complexity and lines required and wrote out a pseudo-implementation for the missing functionality. The newly created reimplementations had a reduced class quality compared to the originals, as the transformation had a significant amount of code in a single file. We improved this by separating the API functionality for control-flow handling and constructors from the actual implementations.

Statistics such as lines of code and complexity are better in our reimplementations than in their original counterparts. Modularity, as calculated through the number of lines and functions per class, assessed that our reimplementation of Cobertura is a bit unreadable due to the amount of code in a single file. Our reimplementation of JaCoCo proved to still be fairly modular. From these statistics, we can further conclude that creating a new idea from scratch would most certainly be easier to do by using QVTo and Mod-BEAM. Transformations made through Mod-BEAM might be easier or harder to maintain depending on how large the transformation becomes. Some suggestions were provided in order to mitigate this problem, such as the use of patterns similar to ASM in order to move code to the back-end of the transformation.

By reasoning about an implementation of the weaving process of AspectJ, we came to the conclusion that it is feasible to make such a product. This product would not share any code except for API-level functionality with the CCA implementations. We also came to the conclusion that Mod-BEAM could still be tested on stability by broadening the scope of test-cases so that it can be confirmed that the tool is completely stable.

- *How reusable is a model-driven solution using Mod-BEAM compared to the original implementation?*

A reusability comparison is performed in Section 5.6.1. In this section, we reason that the reimplementations can be separated into a transformation library for the API, a transformation for the Cobertura logic and a transformation for the JaCoCo logic. We come to the conclusion that there is some overlap in the logic to determine which instruction types should be instrumented. Most of the logic is, however, implementation-dependent and therefore not reusable.

- **What are the benefits and drawbacks when implementing code instrumentation as model transformation through Mod-BEAM when compared to the implementation as code transformation?**

We have drawn various conclusions for each respective case-study (Sections 4.7, 5.7, 6.5). Summarised, we have come to conclude that our reimplementations of each corresponding tool have so far consisted of fewer lines of code and lower complexity than their original counterparts. We trade this in for worse readability as each of our reimplementations consists of a single large file and a separate file for the API logic. We believe however that the lowered complexity far outweighs the reduced readability because of the large difference in lines of code. We briefly investigated the performance of our reimplementations and found them to be faster than Cobertura and JaCoCo. This is, however, only because the .class files were already converted to model objects, and does not take the conversion back to a .class into account.

In conclusion, we have reached the goal of defining how a model-transformation implementation through Mod-BEAM compares to the implementation by the original tool, while also describing how the approaches compare and how this can be measured. We came to conclude the main research question by describing in which aspects our implementations hold the advantage when compared to their original counterparts.

7.2 Future work & Recommendations

This thesis has assessed that Mod-BEAM can be used to reproduce functionality from the CCA tools Cobertura and JaCoCo. Our reimplementation of the aforementioned tools significantly reduces size and complexity, but there is still room for further testing and improvement.

- Further research could be performed to check if Mod-BEAM has any edge-cases in which it breaks. This situation was encountered during the research a couple of times and was fixed extremely fast by the developer.
- The reimplementations of the instrumentation logic of Cobertura and JaCoCo are largely complete, but there are still cases where the results differ (try/catch blocks or multi-condition statements). Support for these cases could be added to create a full recreation of the original counterparts.
- Further study the performance. While this was briefly done in this research, further research can still be done on performance using larger test-samples or by calculating the total time required to generate the model and perform the instrumentation.
- Alternatively, AspectJ could be worked out to prove that even the weaving

process of such a large tool can potentially be simplified into a much more compact implementation.

- Expand upon the bytecode transformation API that was created in QVTo. There are still calls that seem redundant (manually linking instructions, rather than inserting them) and the current version might allow for too much freedom (have to manually handle cases rather than predefined overridable functions such as visitVar, visitJump, visitSwitch, etc.).

- Lastly, search for a more complete hybrid model transformation language. While ETL was used as a potential migration candidate for our QVTo implementation, it proved to be less easy to work with than QVTo, while also missing functionality that our implementation is highly reliant on (constructors).

Bibliography

[1] A. M. Lund, “Measuring usability with the use questionnaire12,” *Usability interface*, vol. 8, no. 2, pp. 3–6, 2001.

[2] A. W. R. Emanuel, R. Wardoyo, J. E. Istiyanto, and K. Mustofa, “Modularity index metrics for java-based open source software projects,” *arXiv preprint arXiv:1309.5689*, 2013.

[3] C. Gerpheide, R. Schiffelers, and A. Serebrenik, “Assessing and improving quality of qvto model transformations,” *Software Quality Journal*, vol. 24, 06 2015.

[4] M. van Amstel, S. Bosems, I. Kurtev, and L. Ferreira Pires, “Performance in model transformations: Experiments with atl and qvt,” 06 2011, pp. 198–212.

[5] J. Cabot, R. Claris, E. Guerra, and J. de Lara, “Verification and validation of declarative model-to-model transformations through invariants,” *Journal of Systems and Software*, vol. 83, no. 2, pp. 283 – 302, 2010, computer Software and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121209001976

[6] K. Lano, S. Kolahdouz Rahimi, and S. Yassipour Tehrani, “Analysis of hybrid model transformation language specifications,” 04 2015.

[7] R. Dvorak, “Model transformation with operational qvt,” *Borland Software Corporation*, vol. 19, 2008.

[8] S. Boyko, R. Dvorak, and A. Igdalov, “The art of model transformation with operational qvt,” 2009.

[9] I. Keselev, *Aspect-oriented programming with AspectJ*. Sams, 2003.

[10] E. Hilsdale and J. Hugunin, “Advice weaving in aspectj,” in *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, ser. AOSD ’04. New York, NY, USA: ACM, 2004, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/976270.976276

Appendix A

Instruction comparison code

```
1  // Has a granularly increasing level of complexity
2  public class InputClass {
3          public static void main(String[] args) {
4                  InputClass a = new InputClass();
5                  InputClass b = new InputClass(-1);
6
7                  a.empty();
8                  a.simplePrint();
9                  a.simpleIf(true);
10                 a.complexIf(50);
11                 a.singleLine();
12                 a.varTest();
13
14                 a.printing();
15                 a.iterating();
16                 a.array();
17
18                 a.switchTest(5);
19
20                 a.amalgamation();
21         }
22
23         public InputClass() {
24                 System.out.println("Default␣constructor");
25         }
26
27         public InputClass(int i) {
28                 System.out.println("Itemized␣constructor:␣" + i);
29         }
30
31         private void empty() {
32
33         }
34
35         private void simplePrint() {
36                 System.out.println("Test!");
37                 System.out.println("Test2!");
38         }
39
40         private void simpleIf(boolean a) {
41                 if(a) {
42                         System.out.println("True!");
```

```
43                  } else {
44                          System.out.println("False!");
45                  }
46          }
47
48          private void complexIf(int i) {
49                  if(i < 10) {
50                          System.out.println("i:␣lower␣or␣equal␣to␣ten!");
51                  } else if(i <= 50) {
52                          System.out.println("i:␣between␣10␣and␣100!");
53                  } else {
54                          System.out.println("i:␣higher␣than␣100!");
55                  }
56          }
57
58          private void singleLine() {
59                  int i = 5; int j = 10; int k = 50;
60          }
61
62          private void varTest() {
63                  int i = 10;
64                  int j = 10;
65
66                  i++;
67                  j++;
68                  i--;
69                  j--;
70
71                  System.out.println("i:␣" + i);
72                  System.out.println("j:␣" + j);
73
74                  System.out.println("A");
75                  System.out.println("B");
76                  System.out.println("C");
77          }
78
79          private void printing() {
80                  System.out.println("A!");
81                  System.out.println("B!");
82                  int i = 10;
83                  System.out.println("C!");
84                  System.out.println("D!");
85
86                  String test = (i < 10) ? "E" : "F";
87                  System.out.println(test);
88          }
89
90          private void iterating() {
91                  for(int j = 0; j < 10; ++j) {
92                          System.out.println("For");
93
94                          int val = j;
95                          while(true) {
96                                  if(val < 0) {
97                                          break;
98                                  }
99
100                                 System.out.println("While");
101
```

```
102                             val--;
103                     }
104             }
105     }
106
107     private void array() {
108             int[] items = new int[] {
109                             5, 10, 30
110             };
111
112             for(int item : items) {
113                     System.out.println(item);
114             }
115     }
116
117     private void switchTest(int param) {
118             switch (param)
119             {
120                     case 1 :
121                             System.out.println("Switch␣1");
122                             break;
123                     case 5 :
124                             System.out.println("Switch␣5");
125                             break;
126                     default :
127                             System.out.println("default");
128                             break;
129             }
130     }
131
132     private void amalgamation() {
133             // Printing
134             System.out.println("A!");
135             System.out.println("B!");
136             int i = 10;
137             System.out.println("C!");
138             System.out.println("D!");
139
140             String test = (i < 10) ? "E" : "F";
141             System.out.println(test);
142
143             // Iterating
144             for(int j = 0; j < 10; ++j) {
145                     System.out.println("For");
146
147                     int val = j;
148                     while(true) {
149                             if(val < 0) {
150                                     break;
151                             }
152
153                             System.out.println("While");
154
155                             val--;
156                     }
157             }
158
159             // Array
160             int[] items = new int[] {
```

```
161                                 5, 10, 30
162                 };
163
164                 for(int item : items) {
165                         System.out.println(item);
166                 }
167         }
168 }
```

Listing A.1: InputClass.java

```
1
2  import java.util.Arrays;
3
4  // Most cases come from recursive implementations
5  // found on the internet
6  public class Recursion {
7      public static void main(String[] args)
8      {
9          long oldTime = System.currentTimeMillis();
10
11         int arr[] = {64, 34, 25, 12, 22, 11, 90};
12
13         bubbleSort(arr, arr.length);
14
15         long newTime = System.nanoTime();
16         float diff = (newTime - oldTime);
17
18         System.out.println("Sorted␣array␣:␣");
19         System.out.println(Arrays.toString(arr));
20         System.out.println("In␣" + diff + "␣ms");
21     }
22
23     // Source: https://www.geeksforgeeks.org/java-program-for-recursive-bubble-sort/
24     public static void bubbleSort(int arr[], int n)
25     {
26         if (n == 1)
27             return;
28
29         for (int i=0; i<n-1; i++)
30             if (arr[i] > arr[i+1])
31             {
32                 // swap arr[i], arr[i+1]
33                 int temp = arr[i];
34                 arr[i] = arr[i+1];
35                 arr[i+1] = temp;
36             }
37
38         bubbleSort(arr, n-1);
39     }
40
41     // Source: https://www.java67.com/2014/07/quicksort-algorithm-in-java-in-place-example.
           html
42     public void quickSort(int[] array) {
43         recursiveQuickSort(array, 0, array.length - 1);
44     }
45
46     public void recursiveQuickSort(int[] array, int startIdx, int endIdx) {
47         int idx = partition(array, startIdx, endIdx);
```

```
48
49          // Recursively call quicksort with left part of the partitioned array
50          if (startIdx < idx - 1) {
51              recursiveQuickSort(array, startIdx, idx - 1);
52          }
53
54          // Recursively call quick sort with right part of the partitioned array
55          if (endIdx > idx) {
56              recursiveQuickSort(array, idx, endIdx);
57          }
58      }
59
60      public int partition(int[] array, int left, int right) {
61          int pivot = array[left];
62
63          while (left <= right) {
64              while (array[left] < pivot) {
65                  left++;
66              }
67
68              while (array[right] > pivot) {
69                  right--;
70              }
71
72              if (left <= right) {
73                  int tmp = array[left];
74                  array[left] = array[right];
75                  array[right] = tmp;
76
77                  left++;
78                  right--;
79              }
80          }
81          return left;
82      }
83
84      // Check if MB generates branches 5 to 150 or only 5 and 150
85      public void switchTest(int param) {
86                  switch (param)
87                  {
88                          case 5 :
89                                  System.out.println("Switch␣1");
90                                  break;
91                          case 150 :
92                                  System.out.println("Switch␣5");
93                                  break;
94                          default :
95                                  System.out.println("default");
96                                  break;
97                  }
98          }
99  }
```

Listing A.2: Recursion.java

```
1   // Most cases come from Cobertura source-code
2   class Recursion {
3           public void simpleMultiCondition2(boolean a, boolean b) {
4                   if(a || b) {
```

```
5                       System.out.println("true");
6               }
7               System.out.println("false");;
8       }
9
10      public void simpleMultiCondition(boolean a, boolean b) {
11              if(a && b) {
12                      System.out.println("true");
13              }
14              System.out.println("false");;
15      }
16
17      public void simpleMultiCondition(boolean a, boolean b, boolean c) {
18              if(a && b || c) {
19                      System.out.println("true");
20              }
21              System.out.println("false");;
22      }
23
24      public void callMultiCondition(int a, int b, int c)
25      {
26              if ((a == b) && (b >= 3) || (c++ < a))
27              {
28                      System.out.println("true");
29              }
30      }
31
32      public void callMultiCondition2(int a, int b, int c)
33      {
34              if ((a == b) && (b >= utilEcho(3)) || (c < a))
35              {
36                      System.out.println("true");
37              }
38      }
39
40      int utilEcho(int number)
41      {
42              return number;
43      }
44  }
```

Listing A.3: Multicondition.java