Reverse Engineering of Java Programs through Static and Dynamic Analysis to generate Scenario Diagrams

By

Bojana Kolbah, B.Eng

A thesis submitted to the Faculty of Graduate Studies and
Postdoctoral Affairs

in partial fulfillment of the requirements for the degree of
**Master of Applied Science in Electrical and Computer
Engineering**

Ottawa-Carleton Institute of Electrical and Computer Engineering
(OCIECE)

Department of Systems and Computer Engineering
Carleton University

Ottawa, Ontario, Canada, K1S 5B6

January 2011

©Copyright 2011, Bojana Kolbah

Library and Archives
Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-79539-2
Our file *Notre référence*
ISBN: 978-0-494-79539-2

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

This thesis builds on previous work towards reverse engineering of code to produce UML sequence diagrams by combining static and dynamic analysis of Java systems to obtain an execution scenario diagram to aid in program comprehension for the purpose of software verification and maintenance. We extract the control flow graph from the system source code and obtain an execution trace by instrumenting and running the system. To store the acquired data, we define two formal models, one for static and one for dynamic analysis. The output is presented with the UML sequence diagram standard notation. The transformation from the two input models to the UML sequence diagram is done through the use of third party model transformation tool.

Our approach was validated on four Java systems including a Library System from previous reverse engineering study. One of the goals of this work is to reduce the execution overhead inherent in dynamic analysis. By combining dynamic with static system data, we found that the overhead, in terms of execution time, was significantly reduced without major impact on the level of program comprehension.

ACKNOWLEDGEMENTS

I wish to sincerely thank Dr. Labiche for his guidance and support for this work.

A great thanks to my family: Chris, Tanja, Slobodan and Barbara who have gone through many pains and efforts to make it possible for me to complete my degree.

Contents

1	Introduction.....	1
2	Related work	4
2.1	Leduc's approach.....	4
2.1.1	Aspects and AspectJ	4
2.1.2	AspectJ for tracing.....	5
2.2	Other related work.....	9
3	Instrumentation and control flow	11
3.1	Introduction	11
3.2	Trace information	13
3.2.1	Running example	13
3.2.2	Logger	17
3.2.3	Object identification.....	18
3.2.4	Method call interception.....	21
3.2.5	Trace file format	26
3.3	Control flow analysis.....	29
4	Trace and control flow models.....	34
4.1	Trace model.....	34
4.1.1	Model elements.....	34
4.1.2	Example instance of the model.....	37
4.2	Control flow model.....	38
4.2.1	Model elements.....	38
4.2.2	Example instance of the control flow model.....	40
4.3	The UML Superstructure (excerpt)	41
5	Mapping the models to the scenario diagram	47
5.1	Linking trace execution logs to source code	48
5.2	Basic sequence diagram elements	48
5.3	Combined Fragments.....	55
5.4	Ordering of send and reply Messages.....	59
5.5	Transformation with MDWorkbench.	60
5.6	Example	63
6	Case study	70
6.1	Research questions.....	70

6.2	The case study systems	70
6.3	Test cases	72
6.3.1	Test case setup	72
6.3.2	Execution overhead.....	73
6.3.3	Execution inputs and repeated executions	73
6.4	Measurement	74
6.5	Experimental results	76
6.5.1	Execution time.....	76
6.5.2	Class size.....	80
6.5.3	Scenario diagram visualization.....	81
7	Conclusion	88
8	Future work.....	89
9	References.....	92
Appendix A	Aspect files for the running example	94
Appendix B	Control flow graphs for the running example	98
Appendix C	Trace and control flow models	100
Appendix D	Trace, control flow graph for the running example.....	103
Appendix E	Model transformation code.....	108

List of figures

Figure 1 – MethodExecutionAspect (excerpt) for the Library System case study [2].....	6
Figure 2 – An example of method instrumented using Leduc’s method	7
Figure 3 – Excerpt of the ControlFlowAspect for Library System case study in [2]	8
Figure 4 – Source code for class A from the running example	14
Figure 5 – Source code for class B from the running example	14
Figure 6 – Source code for class C from the running example	15
Figure 7 – Source code for class D from the running example	15
Figure 8 – Manually generated sequence diagram for the running example	16
Figure 9 – Source code for Logger	18
Figure 10 – Section of the IdentifierAspect.aj relating to class A from the running example.....	19
Figure 11 – Pointcut definitions for the running example.....	22
Figure 12 – Code executed before pointcut callMethod	23
Figure 13 – Code executed before pointcut callStaticMethod	24
Figure 14 – Code executed before pointcut callConstructor	25
Figure 15 – Fragment of Trace.txt contents for the running example	27
Figure 16 – Trace.xmi contents for the running example	28
Figure 17 – Class B control flow graph for the running example	31
Figure 18 – Class B from the running example displayed in control flow graph.....	32
Figure 19 – Trace model.....	35
Figure 20 – Partial representation of the running example trace as trace model instance	37
Figure 21 – Control flow model.....	39
Figure 22 – Partial representation of the control flow model instance for the running example.....	41
Figure 23 – Messages in UML Superstructure [18].....	42
Figure 24 – Lifelines in UML Superstructure [18].....	43
Figure 25 – Occurrence and Execution Specifications in UML Superstructure [18]	44
Figure 26 – Events in UML Superstructure [18]	45
Figure 27 – CombinedFragment in UML Superstructure [18].....	46

Figure 28 – Matching <code>MethodCall:ControlFlow</code> to <code>MethodLog:Trace</code>	48
Figure 29 – Creating unique sequence diagram elements for <code>MessageLogs</code>	49
Figure 30 – Mapping sending <code>Message</code> for <code>MessageLog</code>	50
Figure 31 – Mapping sending <code>Lifeline</code> for <code>MessageLog</code>	51
Figure 32 – Mapping receiving <code>Lifeline</code> for <code>MessageLog</code>	51
Figure 33 – Mapping receiving <code>Class</code> for <code>MessageLog</code>	52
Figure 34 – Mapping sending <code>Class</code> for <code>MessageLog</code>	53
Figure 35 – Mapping sending <code>Actor</code> for <code>MessageLog</code>	53
Figure 36 – Mapping reply <code>Message</code> for <code>MessageLog</code>	54
Figure 37 – Mapping <code>BehaviourExecutionSpecification</code> for <code>MessageLog</code>	54
Figure 38 – Pseudo code for nesting <code>CombinedFragments</code>	57
Figure 39 – Transformation ruleset (excerpt about <code>Model</code> , <code>Collaboration</code> and Interaction).....	61
Figure 40 - Transformation ruleset (excerpt about <code>Lifelines</code>)	62
Figure 41 – Scenario diagram for the running example, part 1	63
Figure 42 – Scenario diagram for the running example, part 2.....	65
Figure 43 – Scenario diagram for the running example, part 3	66
Figure 44 – Scenario diagram for the running example, part 4.....	67
Figure 45 – Scenario diagram for the running example, part 5	68
Figure 46 – Scenario diagram produced with the original tool [2].....	74
Figure 47 – Excerpt of the <code>.bat</code> file used to run experiment.....	75
Figure 48 – Class <code>Time</code>	75
Figure 49 – Excerpt of the experimental output	76
Figure 50 – Execution time vs. instrumentation level by program.....	77
Figure 51 – Median execution time vs. method calls executed.....	79
Figure 52 - Impact of instrumentation on class size	80
Figure 53 – Scenario diagram for <code>Calculator</code> – part 1.....	82
Figure 54 – Scenario diagram for <code>Calculator</code> – part 2.....	83
Figure 55 – Scenario diagram for <code>Add Copy</code> use case, part 1	84
Figure 56 - Scenario diagram for <code>Add Copy</code> use case, part 2.....	85
Figure 57 - Scenario diagram for <code>Add Copy</code> use case, part 3.....	86

Figure 58 – IdentifierAspect.aj	94
Figure 59 – MethodAspect.aj	97
Figure 60 – ObjectID.java	97
Figure 61 – Class A control flow diagram for the running example	98
Figure 62 - Class C control flow diagram for the running example	98
Figure 63 - Class D control flow diagram for the running example.....	99
Figure 64 – Trace model specification.....	100
Figure 65 – Control flow model specification.....	102
Figure 66 – Trace.txt contents for the running example	104
Figure 67 – Trace.xmi contents for the running example	106
Figure 68 – ControlFlow.xmi contents for running example	107
Figure 69 – Transformation ruleset.....	118
Figure 70 – SignatureComparator.java	119

List of tables

Table 1 – Case studies characteristics	71
Table 2 – Number of method calls for each program	77

1 INTRODUCTION

Reverse engineering is a concept with powerful implementations in software verification and maintenance. By reverse engineering a software system, we can obtain a coherent view of the program structure and its run-time behavior. This allows understanding of the System Under Study (SUS) with less effort than by manually examining the source code. This is particularly true for object oriented programs with complex polymorphism and dynamic binding relationships.

Reverse engineered representations of the software system can be used during software verification to analyze the difference between design and implementation. During software maintenance, this representation can assist in locating where changes need to be made and determining how these changes will affect the larger system.

In this work we build on previous research towards reverse engineering code to produce UML sequence diagrams to aid in program comprehension and verification. A combination of static and dynamic analysis of Java code and its execution is used to obtain a representation of a single execution path through the (SUS). This is then represented in a scenario diagram drawn in UML 2.1 sequence diagram format. We define scenario diagram as a visualization using the sequence diagram notation of a single execution trace. In particular, we do not attempt to recognize several iterations of a single loop. This means that iterations of a single loop will be visually represented as a sequence of separate loops. This is to delineate beginning and end of a loop execution as well as to represent execution of loops in a correct order. Because of this, a scenario diagram will never have if and else sections of the same condition shown inside the same innermost construct. This choice reflects the direction of our research of which this work is an intermediate step. Here we strive to represent execution traces correctly while a future goal is to extract system design sequence diagrams from extracted execution traces (i.e, scenario diagrams). Because sequence diagram generation is an overarching goal, and because this format is widespread and well known, we choose to visualize our

scenario diagrams using the UML notation even though loops and if-else conditions are defined differently.

The static analysis consists of mapping the source code into a control flow model, which summarizes method calls and control flow structures within methods and methods within classes. Dynamic analysis is done by instrumenting the source code with AspectJ, a well-known implementation of Aspect Oriented Programming (AOP), and obtaining traces of method calls made during execution. The traces are mapped into a trace model. The obtained static analysis' control flow model and dynamic analysis' Trace Model are then processed using MDWorkbench, an off-the-shelf Model Transformation tool, to obtain an instance of UML 2.1 sequence diagram representing the program execution.

Though there are a number of tools, such as RSA [1], which can reverse engineer source code into sequence diagrams, they limit themselves to static analysis of the code only. For instance, they do not capture the dynamic binding relationships that occur during executions or capture object interactions and therefore produce sequence diagrams that are necessarily incomplete.

Through the combination of static and dynamic approach, a decreased dependence on instrumentation, compared to a previous study, is achieved and we are able to add instrumentation classes to a Java project without modifying any of the SUS's code. Additionally, the number of instrumentation calls made during execution was decreased by more than half resulting in significantly reduced overhead.

To demonstrate the reduction in overhead, a case study consisting of three different Java programs is presented. To analyze the validity of the obtained UML 2.1 sequence diagram representation of the execution, we use a Library System case study for which we have sequence diagrams (generated by a previously published reverse-engineering technique).

The content of this thesis includes the discussion of related work, Chapter 2, particularly Johanne Leduc's thesis [2] on which this work is based. Chapter 2 also introduces some

concepts related to AOP with AspectJ. Chapter 3 discusses the instrumentation approach and source code Control Flow analysis. The two models (control flow and trace) are formally introduced in Chapter 4 and the transformation to the UML 2.1 sequence diagram format from the derived Models is discussed in Chapter 5. Case studies are presented in Chapter 6, followed by conclusions in Chapter 7 and future work in Chapter 8.

The contributions of this thesis are the following:

- I formally defined models for trace and control flow data;
- I precisely specified model transformations in OCL;
- I used third party model transformation tool to implement model transformations;
- My approach outputs scenario diagrams specified in widely used UML 2.1 format, an improvement over previous work which did not generate easily visualized output;
- I performed several case studies;
- I showed with case studies that my approach significantly reduces instrumentation impact (fewer instrumentation statements occurring during execution resulting in faster execution) and overhead compared to previous work due to the use of both static and dynamic analysis.

2 RELATED WORK

Since this thesis is inspired from and tries to improve over Leduc's thesis, we first present her approach in detail in section 2.1. As we rely on the same technology, namely aspects, this section is also an opportunity to introduce important aspect concepts. Other related work is discussed in section 2.2.

2.1 Leduc's approach

In Leduc's thesis [2], the Java code is instrumented to collect traces at run time. Specifically, the following information is collected: method entry and exit; conditions; loops; distributed information; and concurrency related information. In addition, the source code is instrumented to identify control flow structures exercised during execution (a piece of information that AspectJ cannot observe and intercept): see below.

To collect that information aspects, written with the AspectJ language [3], were defined. Section 2.1.1 describes those concepts while section 2.1.2 discusses how Leduc used them.

2.1.1 Aspects and AspectJ

AspectJ is an implementation of Aspect Oriented Programming (AOP) for the Java language built as an extension to Java [3].

AOP is a paradigm that allows for separation of concerns, which in turn allows for modular code, while at the same time providing a mechanism for the description of concerns that crosscut modules. In [2] and this work, AOP allows us to leave the SUS as one concern while enabling us to crosscut trace instrumentation into the SUS to capture object interaction information.

The following are some concepts used in AOP, and AspectJ in particular.

A **join point** is a well-defined location within the primary code (i.e., SUS) where a concern will crosscut the application.

An **advice** contains the code that is executed when a pointcut is reached. In most AOP systems advice code can execute **before**, **around** or **after** a join point. For a **call**, the program can be interrupted immediately **before**, and/or **after**, the method is called. For **execution**, the interruption happens **before** the first statement or **after** the last statement of the executed method.

Pointcuts allow for groupings of join points. When a pointcut operation is matched, the corresponding advice can be executed. A pointcut uses a designator to specify when a join point is matched. For example, in AspectJ a join point describing a method can be matched on **execution** or **call**.

Aspects are used to group crosscutting concerns and detangle code. They are similar to classes in that they allow for complete encapsulation of code related to a particular concern.

Inter-type declarations are alterations to classes and inheritance hierarchies from outside of the original SUS. For instance, one can use an aspect to add new methods, constructors or fields to a class or declare that a class extends a new superclass. Inter-type declarations affect static-type hierarchy of programs.

2.1.2 AspectJ for tracing

In Leduc's work, several of the concepts above are used to collect trace information.

First, inter-type declarations are used so that each instrumented class extends a specific interface that provides mechanisms to assign each instance a unique object ID. Aspects can then be used to report on object interactions: specifically aspects report on who (ID of caller) calls who (ID of callee).

To capture method calls, two pointcuts are used in [2]. The first one crosscuts method executions, while the second crosscuts object creation. These pointcuts use **around** advices as it is necessary to know when a method execution starts and when it completes.

```

1      Object around(Object self) : (
2          execution(* Employee.EmployeeControlIFFacade.* (..))
3              || execution(* server.EmployeeControlIF.* (..)))
4          && !execution(void Runnable +.run())
5          && !within(Instrumentation.*)
6          && this(self)
7      {
8          ArrayList log = new ArrayList();
9          log.add("Method start");
10         log.add(String.valueOf(((ObjectID) self).getObjectID()));
11         log.add(self.getClass().getName());
12         log.add(thisJoinPointStaticPart.getSignature().
13             toLongString());
13         log.add(thisJoinPointStaticPart.getSignature().getName());
14         LoggingClient.getLoggingClient().instrument(log,
15             thisJoinPoint.getArgs());
15         Object[] returnArg = { proceed(self)};
16         log.clear();
17         log.add("Method end");
18         LoggingClient.getLoggingClient().instrument(log,
19             returnArg);
19         return returnArg[0];
20     }

```

Figure 1 – MethodExecutionAspect (excerpt) for the Library System case study [2]

Figure 1 displays an excerpt of the MethodExecutionAspect used in [2] for a subsystem of the Library System case study. Lines 1 and 2 specify **around execution** advice. This particular pointcut specifies executions by instances of the EmployeeControl interface on the server and client sides (lines 2 and 3) and excludes starting points of threads or any execution within the Instrumentation package (lines 4 and 5). The Instrumentation package contains aspects and related instrumentation files. Logging is done by passing the array list with step-specific information and method arguments to `instrument()` of the LoggingClient. Logging is done before and after the `proceed(self)` statement. Once the method from EmployeeControl interface class is invoked, lines 8 through 14 of Figure 1 will be executed, followed by the execution of the actual method, followed by the execution of lines 16 through 19. Logged information includes interception point (for example “Method start” from line 9 or

“Method end” from line 17), object identification (line 10), class name (line 11), method name (lines 12 and 13), arguments passed to the method (line 14) and return arguments (line 18).

```
1 public void ifAndFor ( ) {
2     if ( attr == 0 ) {
3         Dummy.ifStatementStart( "if ( attr == 0 )", "attr == 0");{
4             D d = new D ( ) ;
5             for ( int i = 0 ; i < totalLoops ; i ++ ) {
6                 Dummy.forStatementStart("for ( int i = 0 ; i < totalLoops
7                     ; i ++ )", "int i = 0", "i < totalLoops", "i ++"); {
8                     if ( i == 1 ) {
9                         Dummy.ifStatementStart( "if ( i == 1 )", "i == 1");{
10                             getD ( d ) ;
11                         }
12                     }
13                     Dummy.ifStatementEnd();
14                     else {
15                         Dummy.ifStatementStart( "else", "!(i == 1)");{
16                             attr = callToA ( ) ;
17                         }
18                     }
19                     Dummy.ifStatementEnd();
20                 }
21             }
22         }
23     }
24 }
25 }
```

Figure 2 – An example of method instrumented using Leduc’s method

AspectJ does not provide a mechanism to instrument control flow structures such as conditions and loops so in order to obtain trace of execution passing through conditional structures—which is necessary to obtain accurate sequence diagrams—Leduc inserted instrumentation statements gathering the information on the type of conditional statement and its parameters directly into the SUS source code. The instrumentation statements are inserted at the beginning and end of each condition or loop as shown in Figure 2. Figure 2 shows a method that has been modified by inserting instrumentation statements (bolded) needed to capture control flow elements. This method is from class B, a class in our running example. Unmodified class B is shown in Figure 5. During system execution the calls to these inserted statements are then detected by the `ControlFlowAspect`, partially shown in Figure 3, and written into the trace.

```

1    before(String statement, String clause) : call(public void
    ifStatementStart(..))
    && args(statement, clause)
2    {
3        ArrayList log = new ArrayList();
4        log.add("If start");
5        log.add(statement);
6        log.add(clause);
7        LoggingClient.getLoggingClient().instrument(log, null);
8    }
9
10   before() : call(public void ifStatementEnd(..))
11   {
12       ArrayList log = new ArrayList();
13       log.add("If end");
14       LoggingClient.getLoggingClient().instrument(log, null);
15   }

```

Figure 3 – Excerpt of the ControlFlowAspect for Library System case study in [2]

ControlFlowAspect specifies a **before call** pointcut. On intercepting an `ifStatementStart(..)`, such as statements on lines 3, 8 and 14 in Figure 2, lines 3 through 7 in Figure 3 are executed. Likewise, interception of `ifStatementEnd(..)` such as 11, 17 and 23 in Figure 2, will cause execution of lines 12 through 14 in Figure 3. In the resulting trace file any method execution traces executing as a result of a condition being true will be delineated by start and end statements produced by this aspect.

Leduc's work covers concurrent and distributed executions. To be able to distinguish trace statements from different execution threads and separate nodes, each instrumentation statement contains an ID unique to each thread and another ID unique to each node. Leduc's approach is able to abstract away from concurrent or distributed communications. Specifically, details about RMI communications are abstracted into asynchronous messages in the UML sequence diagram; details about asynchronous communications between threads through shared data structures are abstracted into asynchronous messages in the UML sequence diagram.

The dynamic analysis presented in [2] involves two instrumentation calls for every method execution and two additional instrumentation calls for each control flow structure (i.e., condition or loop), and requires the instrumentation of the source code (not only the

bytecode). We consider this a heavy instrumentation approach: the execution overhead is large. We believe there is room for improvements, specifically in relation to the way synchronous messages (i.e., calls) and control flow structures are intercepted. Our approach attempts to improve those aspects, leaving aside the reverse engineering of concurrent and (RMI) distributed communications for which Leduc's approach is deemed sufficient: our approach and Leduc's approach to abstract away from concurrent and RMI details would be easy to combine.

2.2 Other related work

The area of program comprehension through dynamic analysis is varied and vibrant as the systematic survey [4] conducted in 2008 by Cornelissen et al. suggests. To get an understanding of studies similar to this one, conducted since Leduc's thesis, aside from the usual searches through IEEE and ACM databases we looked at papers referenced in [4] published since 2003. To augment this, we examined the proceedings for the following conferences:

- International Conference on Software Maintenance (ICSM) [5]
- European Conference on Software Maintenance and Reengineering (CSMR) [6]
- Model Driven Engineering Languages And Systems (MoDELS) [7]
- IEEE International Conference on Program Comprehension (ICPC) [8]

From this analysis, only three pieces of work are related to our approach, in the sense that some kind of static analysis is combined to some kind of dynamic analysis.

Malloy and Power [9] present a technique that combines static and dynamic analysis. First, static analysis is completed by creating a class diagram and a call graph from which the user can select the elements of the system to be instrumented and analyzed dynamically. The call graph consists of nodes representing call sites or functions and edges representing function invocations. The call graph concept is similar to the control flow model presented in this work, but used in a very different way. By selecting only a subset of the SUS from the class diagram to be instrumented with the presented tool, the

user restricts the size of execution trace and resulting sequence and communication diagrams. Also, our control flow model includes flow of control due to conditional statements and loops, which is not the case in Malloy and Power. The remaining part of their analysis is purely dynamic. The instrumentation is done using AspectC++, an AOP implementation for C++ applications with pointcuts specified for object creation and deletion as well as method calls and returns. (Again, control flow is not collected, neither statically as we do, nor dynamically as Leduc did.) The scenario and communication diagram are created in real time from the resulting trace. There are currently no provisions for loops or conditions, nor are they represented in the resulting diagrams.

Cleve and Hainaut [10] conduct dynamic analysis of SQL statements for Data-Intensive applications. AOP statements for SQL are used in a very similar way as we do with AspectJ in this work, particularly in terms of intercepting execution calls and obtaining static information such as the class name, line of code and the object ID for each log statement. Unlike our approach, the trace obtained is analyzed manually only.

Alalfi et al. [11] conduct a study in reverse engineering of UML sequence diagrams for dynamic web applications. Sequence diagrams are generated directly from the trace through purely dynamic means. The traces are trimmed by rejecting any new trace that is identical to a trace existing in the trace database. The authors suggest a similar approach to recognize loops but defer it to future work.

Hamou-Lhadj's PhD thesis [12] analyses large execution traces and proposes a Compact Trace Format (CTF) as a standard. As an initial step, we decided not to use this CTF in our instrumentation approach. This should be considered as an improvement in future work.

A search on the Internet can suggest that a number of tools with similar goals as the one produced here exist. We have already mentioned as RSA. MaintainJ [13], reverseJava [14], JSonde [15], javacalltracer [16] and others advertise dynamic sequence diagram generation.

3 INSTRUMENTATION AND CONTROL FLOW

To reduce the amount of information collected at runtime we devise an approach that relies both on dynamic (runtime) and static analysis. The dynamic collection of information relies on aspects (sections 3.2) while the static analysis is about creating a control flow graph (section 3.3). Section 3.1 discusses how the two techniques are complementary in terms of the information they collect.

3.1 Introduction

The premise of this work is to provide a lighter instrumentation strategy than previous work, i.e., Leduc's thesis [2]. We therefore want to (1) avoid instrumenting control flow structures in the source code and (2) limit the impact of aspects, i.e., reduce the number of instrumentation calls made during execution.

To avoid instrumenting the source code, since AspectJ still does not provide pointcuts for control flow structures, we turn to static analysis, in particular a control flow graph created by parsing the source code.

When introducing the static analysis we need to have a way to connect static information to dynamic trace. For example, we can do this by using the class name and method signature already collected from the trace in [2]. There would be cases, however, when this strategy would not suffice. If the method executing has two sequential calls to another method, one inside a condition or a loop and the other one either just outside or inside another condition, this technique would not allow us to determine from where in the currently executing method the method calls were made and therefore it would be impossible to determine if the called method execution should be inside what condition in the scenario diagram. To avoid this problem, instead of instrumenting method execution, we instrument method call and capture the line number and the source file name from where the method call was made. This information will allow us to correctly link dynamic and static data.

If we can associate a method call from the dynamic trace to the location in the source code where that call is made, then the static analysis allows us to determine from which method in which class the call is made and whether it is inside a condition or a loop. Having obtained this location information statically, we no longer need to extract it through the trace: Leduc relied on a combination of control flow instrumentation and dedicated aspects to do the same.

Furthermore, when making a **call** joinpoint, AspectJ yields information about both the source and destination methods which is crucial in determining when a method has finished executing. This will allow us to not have to rely on the “Method end” instrumentation call used by Leduc in [2]. By relying on static analysis, we are able to reduce the number of instrumentation calls, an improvement we expect to translate into significantly lower overhead and faster execution time.

In order to display the scenario diagrams in UML 2.1 format, we need to collect data over and above the information collected by Leduc. We collect method call sort explicitly because it is required by UML 2.1 Message specification. This information is not collected explicitly in [2] though each trace entry indicates whether the method call was “create method”, “method” or “static method” from which this same information can be gleamed. While the target object ID is collected, unique source object ID is not collected in [2] because it is outside of the execution joint point scope. Instead, Leduc gathers this information from dynamically collected method to method call relationship. In short, we obtain method call statement to method call execution relationships by processing method call statement, condition and loop information from static analysis and actual method call executions thanks to a simplified instrumentation join point selection, information that is roughly equivalent information to Leduc’s, and we are then able to construct representation of the execution as was done in [2].

3.2 Trace information

The tracing instrumentation presented here produces a log file with method call and object creation information in a sequential order, as they are executed. Additionally, when an object is created, its class name and unique object ID are written into the trace.

The code for the tracing instrumentation entails the following four files: `ObjectID.java`, `Logger.java`, `MethodAspect.aj`, and `IdentifierAspect.aj`. These files are specific to the SUS as they contain package and class names from that system, i.e., the classes whose behavior needs to be intercepted. These need to be created anew for each SUS, but creating them is easy to automate (both Leduc's and our approach generate these files automatically). They provide three functionalities: logging, object identification and method call interception; which are discussed in sections 3.2.2, 3.2.3, and 3.2.4 respectively. Section 3.2.5 then presents the format of the file generated by the logger. A running example is used all along the chapter to illustrate the approach: it is first presented in section 3.2.1; the example of tracing instrumentation can be found in Appendix A.

3.2.1 Running example

Throughout this dissertation the following example will be used to demonstrate implementation details. This example, shown in its entirety in Figure 4 through to Figure 7 is an abstract example written to demonstrate the main features of our approach. For that purpose the example contains several conditions and loops, some of them nested. Additionally, we have instances of dynamic binding to a method in a parent class.

Class A (Figure 4) contains methods whose representation in UML sequence diagram will be different. `EmptyCall` and `SimpleMultiplication` do not make calls to any other methods that will be visible in UML representation. `ThreeCalls` method will be shown to make three calls, two to the object on which it is executing and one to object b. The constructor will make one call visible in the resulting diagram.

```

1  package example;
2
3  public class A {
4      public int attr;
5      public void emptyCall(String str) {}
6      public void simpleMultiplication(int x) {attr = -1 *x;}
7      public int threeCalls(B b) {
8          emptyCall("arg");
9          if(b.attr > 0){
10             simpleMultiplication(b.attr);
11         }
12         b.quickCall();
13         return 5;
14     }
15     public A(){
16         simpleMultiplication(10);
17     }
18 }

```

Figure 4 – Source code for class A from the running example

```

1  package example;
2
3  public class B {
4      private int totalLoops = 0;
5      public int attr = 0;
6      public A a = new A();
7      public B(int max){ totalLoops = max;}
8      public void ifAndFor() {
9          if (attr == 0){
10             D d = new D();
11             for(int i = 0; i < totalLoops; i++){
12                 if(i == 1){
13                     getD(d);
14                 }
15                 else {
16                     attr = callToA();
17                 }
18             }
19         }
20     }
21     public void getD(D d) {attr = d.get();}
22     public int callToA() {return a.threeCalls(this);}
23     public void quickCall() {}
24 }

```

Figure 5 – Source code for class B from the running example

Class B (Figure 5) contains nested conditions including loop and if-else option. Object d is created inside of the outermost loop while calls `getD()` and `callToA()` should be presented inside several nested control flow structures.


```
1  package example;
2
3  public class C extends B{
4      public C(int max){ super(max);}
5      public static void main(String[] arg) {
6          C c = new C(Integer.parseInt(arg[0]));
7          c.ifAndFor();
8      }
9  }
```

Figure 6 – Source code for class C from the running example

Class C (Figure 6), inherits from B and, as such, it will be used to demonstrate the handling of dynamic binding. This class also contains the main method of the program with the argument indicating how many times a loop in B will execute, a key feature in making this example useful for evaluating execution time later in this work.

```
1  package example;
2
3  public class D {
4      int d = 0;
5      public void set(int x){ d = x;}
6      public int get () {return d;}
7  }
```

Figure 7 – Source code for class D from the running example

Class D (Figure 7), is a very simple class called by class B.

Figure 8 – Manually generated sequence diagram for the running example

Figure 8 contains a manually generated sequence diagram for the running example code starting from the main method in class `C`. This is a true sequence diagram in the sense that it contains alternate paths in the same view. This contrasts the scenario diagrams we will generate for this example, which will represent traversals of the loop, corresponding to different executions (test cases) separately.

The features of the running example are displayed in the diagram. During the construction of `C`, `A` is created. `A`'s constructor is then executed. Both construction of `A` and method `ifAndFor()` are examples of `C` dynamically binding to `B`'s methods. The three nested loops from `ifAndFor()` are shown including the `alt` construct.

3.2.2 Logger

Class `Logger` (Figure 9) implements the logging functionality. The `Logger` of [2] has been significantly modified: it is much simpler since multithreading and RMI is outside of scope for this work. Specifically, unlike the `Logger` in [2], we do not need to retain the timestamp of the trace statement, node on the network or thread identifications.

The `Logger` implements the singleton design pattern. All trace statements generated by the aspects (see following sections) are written into a single file, `Trace.txt`. To write into `Trace.txt` `instrument(List <String> record)` is called which will write all `String` statements from the `record` into the file. The `Logger` needs to be generated for every project to be instrumented. However, with exception of the package specification on the first line, all `Logger` code is generic.

```

1
2  public class Logger {
3      private static Logger instance = new Logger();
4      private FileWriter filewriter = null;
5      public static Logger getLoggingClient() {
6          return instance;
7      }
8      private FileWriter getFileWriter() {
9          try {
10             if (filewriter == null) {
11                 filewriter = new FileWriter(new File("Trace.txt"));
12             }
13             } catch (IOException e) {
14                 System.out.println("Error in Logging Client: "+e);
15             }
16             return filewriter;
17         }
18     public void instrument(List <String> record) {
19         FileWriter writer = getFileWriter();
20         try {
21             for (int count = 0; count < record.size(); count++){
22                 writer.write((String) record.get(count));
23                 writer.write("\r\n");
24             }
25             writer.flush();
26             } catch (IOException e) {
27                 System.out.println("Error in Logging Client: "+e);
28             }
29         }
30     }

```

Figure 9 – Source code for Logger

3.2.3 Object identification

The IdentifierAspect aspect (Figure 10) along with the ObjectID interface are used to correctly set and make available to other aspects a unique identifier and class name for objects whose behavior is to be monitored/traced. The ObjectID interface simply specifies one method called getObjectID() that returns an integer: a unique identifier for the instance of the class on which it is called. Two instances of the same class cannot have the same identifier (this is ensured by the IdentifierAspect aspect), while instances of different classes can have the same identifier. Instances of a class are therefore uniquely identified using their identifier, while instances in the SUS, possibly from different classes, are uniquely identified using the unique pair (class name, identifier). Giving a unique identifier to an instance of a class is triggered during

construction, as discussed in section 3.2.4. The `ObjectID` interface code is generic, with the exception of the package name. This file is therefore generated for each project.

```
1  private int A.objectID = A.objectIDgenerator(objectID);
2  private static int A.currentObjectID = 1;
3  private static int A.objectIDgenerator(int i) {
4      int id = i;
5      if (i < 1){
6          LinkedList <String> log = new LinkedList <String> ();
7          id = A.currentObjectID++;
8          log.add("<lifeline className=\"example.A\" name=\"A_\"
9              + id + \"\"/>");
10         Logger.getLogger().instrument(log);
11     }
12     return id;
13 }
14 declare parents : A implements ObjectID;
15 public String A.getObjectID() {
16     if (objectID < 1){
17         objectID = A.objectIDgenerator(objectID);
18     }
19     return "A_" + objectID;
20 }
```

Figure 10 – Section of the IdentifierAspect.aj relating to class A from the running example

Instead of discussing a generic aspect, we decided to discuss the IdentifierAspect aspect by means of an example. Figure 10 contains the aspect code to instrument class A from the running example. IdentifierAspect.aj needs to contain a separate section, similar to the one of Figure 10 for class A, for each instrumented SUS class. To create this section for another class, the programmer needs to replace string A in Figure 10 with the appropriate class name, which can easily be done automatically. As it contains code presented in Figure 10 for each instrumented class as well as the package name at the top of the file, the IdentifierAspect.aj file is generated for each project.

The aspect adds an attribute of type `int`, named `objectID` to the instrumented object (line 1). It also specifies that the class (in this case class A) implements a new interface, specifically `ObjectID` (line 13), and adds an implementation to the method declared in this interface, specifically `getObjectID()` (lines 14-19).

The aspect also adds to the class (static attribute `currentObjectID` and method `objectIDgenerator()`) the capability to count the number of its instances, which is the mechanism used to set a unique ID to those instances. Attribute `objectID` is obtained for each object of the instrumented class during its creation by calling static method `objectIDgenerator(objectID)`. `objectIDgenerator(objectID)` will initialize attribute `objectID` by incrementing static attribute `currentObjectID` (line 7). At this time, a call will be made to the `Logger` to record that this object has just been created (lines 8-9), since the first time this method is called is during construction. `Logger` will record the line with placeholder keyword “lifeline”. The word “lifeline” is used because an object executing a method will eventually be represented as a lifeline in the sequence diagram with the object ID from this line as its identifier. During the lifetime of this object, the object’s unique ID, that appears in the trace as the class name followed by the underscore character and the `objectID` numeral (line 8), can be accessed by calling method `getObjectID()`, specified by the `ObjectID` interface.

If the object created is of a class that has parent classes and those classes have a behavior that is also intercepted, the `objectGenerator` method will be called for each parent and then for the child, following the order in which constructors in an inheritance hierarchy are called in Java. This will cause `Logger`’s `instrument()` method to be called for each of these in sequential order from parent to child. If the parent class has a constructor method, it will be called before the child’s `objectID` attribute is set by the `objectIDGenerator` method. If then the parent’s constructor contains method calls, the child’s `objectID` would be read by the aspect before it is initialized to its final value, as specified in `objectIDgenerator()`. The trace file would then have incorrect information pertaining to the ID of the object: calls in the parent constructor to methods overridden in the child would lead to log entries with the wrong object identifier. To avoid this, whenever `getObjectID()` is called, a check is made whether the `objectID` attribute has been initialized by checking whether its value is less than 1 (line 15). If the initialization has not happened, initialization is performed before `getObjectID()` returns. To avoid overwriting this value at actual initialization, the

`objectIDgenerator()` verifies (line 5) whether the `objectID` has already been initialized and does not modify it if this is the case. Since `objectIDgenerator()` is static, it cannot access the `objectID` attribute of the instance. This information therefore has to be passed as a parameter: lines 16 and 1.

Note that in Leduc's thesis, the identifier aspect did not include any calls to the `Logger`, yet here we log the sequence in which the object IDs are created including IDs for parent objects which will never be used in our analysis. Detailed explanation for this will be given in constructor interception section of Section 3.2.4.

3.2.4 Method call interception

`MethodAspect.aj` defines pointcuts and aspects that are executed at these pointcuts. The pointcuts are defined to intercept method calls (either to instance methods or static methods) and perform tasks before them, and to intercept calls to constructors and perform tasks before them.

Figure 11 shows the pointcut definitions for the running example. Pointcut `callMethod` specifies all method calls (line 1), i.e., with any method name and any list of parameters, except for static method calls (line 2) and calls made to `IdentifierAspect`, `Logger` or `ObjectID` classes/aspects (lines 3-5). Likewise, pointcut `callStaticMethod` specifies all calls to static methods (line 6) that are made to the SUS but not to our instrumentation (lines 7-10). Finally, pointcut `callConstructor` specifies calls to constructors of the SUS (line 11), omitting calls to constructors in the instrumentation infrastructure (line 12). Note that the aspect classes and interface (`IdentifierAspect`, `Logger` and `ObjectID`) are supposed to be part of the instrumented package (here example), thereby assuming that the instrumented system does not already have these classes and interfaces. If this is the case, similarly to Leduc's work, aspect classes and interface can easily be put in a dedicated package. We have instead opted for simplicity here.

```

1  pointcut callMethod() : call (* *.*.*(..))
2                                && !call (static * *.*.*(..))
3                                && !call (* example.IdentifierAspect.*(..))
4                                && !call (* example.Logger.*(..))
5                                && !call (* example.ObjectID.*(..));

6  pointcut callStaticMethod() : call (static * *.*.*(..))
7                                && !call (* example.MethodAspect.*(..))
8                                && !call (* example.IdentifierAspect.*(..))
9                                && !call (* *.*.objectIDgenerator(..))
10                               && !call (* example.Logger.*(..));

11 pointcut callConstructor() : call (*.*.new(..))
12                               && !call (example.Logger.new(..));

```

Figure 11 – Pointcut definitions for the running example

All the advices for these pointcuts are executed **before calls**, rather than **executions** as in [2], because we want to know the location of calls in the source code as was discussed in section 3.1. An **execution** join point on the other hand is only aware of the location in the source code of the method being called and not where the call is made from. Recall that we can deduce the source code location of the executed method through static analysis by matching its signature, given that we know the type of the object executing the method, hence we do not need another join point that would provide such information. When defining aspects, a **before** rather than an **after** advice is used because we want to obtain the sequential order in which method calls are made: an after advice would provide the reverse order (e.g., log entry for the callee before a log entry for the caller).

The aspect code executed at the pointcut obtains the information specified in section 3.1. Figure 12 presents the code, executed right before a `callMethod` pointcut, thereby obtaining information about a call to a method of an object, and sending that information to the Logger. Line 2 defines a local variable, `thisID`, in which the ID of the object making the call will be stored. Line 3 defines a local variable to store log information before sending it to the Logger (line 19). Lines 4-9 retrieve the identity of the object performing the call: either it is an object whose behavior is instrumented and implements the `ObjectID` interface, and then we retrieve the object identifier (line 5), or the calling context is static, for example the main method of the program, or a class that has not been

instrumented (line 8). Method `getStaticClassName(...)` will format the passed class file name into format “`classname_static`”.

```
1  before(): callMethod () {
2      String thisID = new String ();
3      LinkedList <String> log = new LinkedList <String> ();
4      if (thisJoinPoint.getThis() != null) {
5          thisID = String.valueOf(((ObjectID)
6              thisJoinPoint.getThis()).getObjectID());
7      }
8      else {
9          thisID = getStaticClassName(thisJoinPointStaticPart.
10              getSourceLocation().toString());
11      }
12      String targetID = new String ();
13      targetID = String.valueOf(((ObjectID)
14          thisJoinPoint.getTarget()).getObjectID());
15      log.add("<messageLog bindToClass=\"" +
16          MethodAspect.getBindToClassName(
17              thisJoinPoint.getTarget().toString()) + "\"
18          messageSort=\"synchCall\" signature=\"" +
19          MethodAspect.getMethodSignature(thisJoinPoint.toString())
20          + "\">");
21      log.add("<sendEvent covered=\"" + thisID + "\"/>");
22      log.add("<receiveEvent covered=\"" + targetID + "\"/>");
23      log.add("<sentFrom lineNumber=\"" + MethodAspect.getLineNumber(
24          thisJoinPointStaticPart.getSourceLocation().toString()) +
25          "\" name=\"" + MethodAspect.getFileName(
26          thisJoinPointStaticPart.getSourceLocation().toString()) +
27          "\"/>");
28      log.add("</messageLog>");
29      Logger.getLoggingClient().instrument(log);
30  }
```

Figure 12 – Code executed before pointcut callMethod

Next, the advice retrieves the unique identifier of the object being called (line 11): `thisJoinPoint.getTarget()` and stores it in the local variable `targetID`. Then the aspect retrieves details on the method being called: class name (line 12, thanks to `getBindToClassName`), method signature (line 12, thanks to `getMethodSignature`), line number where the call is located (line 17, thanks to `getLineNumber`). Note that `thisJoinPoint` returns the signature of the method being called and `thisJoinPointStaticPart.getSourceLocation()` is used to identify the class file name and line number of the method call.

The aspect for pointcut `callStaticMethod` is very similar (Figure 13), except that we do not have any target object identifier to report on. These differences can be observed on line 10 in Figure 12 which does not appear in Figure 13, and line 16 in Figure 12 referring to `targetID` versus the line 12 in Figure 13 which uses method `getStaticLifelineName(.)` to format the passed join point class name into format “`classname_static`”. Currently the case of an instrumented class making a call to an object of an uninstrumented class is not supported. In the future the approach can be modified by either guiding the selection of classes to be instrumented as is done in [9] or by modifying the aspect to handle the case.

```

1  before(): callStaticMethod () {
2      String thisID = new String ();
3      LinkedList <String> log = new LinkedList <String> ();
4      if (thisJoinPoint.getThis() != null) {
5          thisID = String.valueOf(((ObjectID)
6              thisJoinPoint.getThis()).getObjectID());
7      }
8      else {
9          thisID = getStaticClassName(thisJoinPointStaticPart.
10             getSourceLocation().toString());
11      }
12      log.add("<messageLog bindToClass=\"" +
13          MethodAspect.getStaticBindToClassName(thisJoinPoint.toString()
14             ()) + "\" messageSort=\"synchCall\" signature=\"" +
15             MethodAspect.getMethodSignature(thisJoinPoint.toString()) +
16             "\">");
17      log.add(" <sendEvent covered=\"" + thisID + "\"/>");
18      log.add(" <receiveEvent covered=\"" + getStaticLifelineName(
19          thisJoinPoint.toString()) + "\"/>");
20      log.add(" <sentFrom lineNumber=\"" +
21          MethodAspect.getLineNumber(thisJoinPointStaticPart.getSource
22             Location().toString()) + "\" name=\"" +
23             MethodAspect.getFileName(thisJoinPointStaticPart.getSourceLo
24             cation().toString()) + "\"/>");
25      log.add("</messageLog>");
26      Logger.getLoggingClient().instrument(log);
27  }

```

Figure 13 – Code executed before pointcut `callStaticMethod`

The aspect to report on calls to constructors (Figure 14), i.e., `callConstructor`, defines local variables to store the `objectID` of the calling object on line 2 and log information on line 3. As is the case in Figure 12 and Figure 13, lines 4-9 retrieve the

identity of the object performing the call. Finally the trace message is formatted on lines 10 through 14 and sent to Logger on line 15.

```
1  before(): callConstructor () {
2      String thisID = new String ();
3      LinkedList log = new LinkedList ();
4      if (thisJoinPoint.getThis() != null) {
5          thisID = String.valueOf(((ObjectID)
6              thisJoinPoint.getThis()).getObjectID());
7      }
8      else {
9          thisID = getStaticClassName(thisJoinPointStaticPart.
10              getSourceLocation().toString());
11      }
12      log.add("<messageLog bindToClass=\""+
13          MethodAspect.getNewBindToClassName(thisJoinPoint.toString())
14          + "\" messageSort=\"createMessage\" signature=\"new \"
15          + MethodAspect.getMethodSignature(thisJoinPoint.toString())
16          + "\">");
17      log.add(" <sendEvent covered=\""+ thisID + "\"/>");
18      log.add(" <receiveEvent covered=\"nothing\"/>");
19      log.add(" <sentFrom lineNumber=\""+ MethodAspect.
20          getLineNumber(thisJoinPointStaticPart.getSourceLocation().
21          toString()) + "\" name=\""+
22          MethodAspect.getFileName(thisJoinPointStaticPart.
23          getSourceLocation().toString()) + "\"/>");
24      log.add("</messageLog>");
25      Logger.getLoggingClient().instrument(log);
26  }
```

Figure 14 – Code executed before pointcut callConstructor

Note that on line 12 the word “nothing” is used in place of the target object ID. This is because the object has not been entirely created at the point where the advice is executed (i.e., right before the call to the constructor is made). In order to be able to construct a sequence diagram, the ID of the created object has to be known. To find this information, the trace entry made by the Identifier aspect discussed in section 3.2.3 is used. To use this entry a post processing of the trace needs to be made and this is discussed in section 3.2.5. We have considered other ways of obtaining this information but since we require the correct order in which the constructors are called, their physical location in the code and the target object ID, with current AspectJ technology we were forced to use call join point to obtain the first two and an instrumentation call from the Identifier aspect to obtain the last item. The drawback of this approach is that the creation of object ID for the object’s parents is also instrumented and has to be removed from the trace in a post

processing step. The instrumentation of parent's ID adds a slight overhead, however, we judged it to be small enough to use this approach until a significantly better one is available. Additionally, post processing the trace is very simple (see section 3.2.5), is performed off-line, and has therefore no impact on the execution of the SUS.

`MethodAspect.aj` is generic with the exception of the package name at the top of the file as well as pointcut definitions. Because of this, the `MethodAspect.aj` file has to be generated for each project.

Note that the code discussed here could have been written in a more elegant format using common software engineering practices such as separation of concerns. While this is definitely what we recommend for the future implementations of this tool, we decided to keep a coding style as similar as possible to Leduc's original coding style and as simple as possible not to incur any additional execution overhead and keep comparisons to Leduc's program simple.

3.2.5 Trace file format

The trace file is written to whenever `MethodAspect` or `IdentifierAspect` make a call to `Logger`. This information is the input to the Model Transformation tool and it needs to be modified to ensure that data is correct and the format complies with the trace model XMI file.

```

1  <messageLog bindToClass="example.C"
    messageSort="createMessage" signature="new example.C(int)">
2      <sendEvent covered="C_static"/>
3      <receiveEvent covered="nothing"/>
4      <sentFrom lineNumber="6" name="C.java"/>
5  </messageLog>
6  <lifeline className="example.B" name="B_1"/>
7  <lifeline className="example.C" name="C_1"/>
8  <messageLog bindToClass="example.A"
    messageSort="createMessage" signature="new example.A()">
9      <sendEvent covered="C_1"/>
10     <receiveEvent covered="nothing"/>
11     <sentFrom lineNumber="6" name="B.java"/>
12 </messageLog>
13 <lifeline className="example.A" name="A_1"/>
14 <messageLog bindToClass="example.A" messageSort="synchCall"
    signature="void example.A.simpleMultiplication(int)">
15     <sendEvent covered="A_1"/>
16     <receiveEvent covered="A_1"/>
17 <sentFrom lineNumber="16" name="A.java"/>
18 </messageLog>
19 <messageLog bindToClass="example.C" messageSort="synchCall"
    signature="void example.C.ifAndFor()">
20     <sendEvent covered="C_static"/>
21     <receiveEvent covered="C_1"/>
22     <sentFrom lineNumber="7" name="C.java"/>
23 </messageLog>
(....)

```

Figure 15 – Fragment of Trace.txt contents for the running example

Trace.txt, as produced by our tracing instrumentation for the running example is partially shown in Figure 15: See Appendix D for the complete file. Lines 1 through 5 are a single trace entry for the interception of a constructor call corresponding to line 5 in Figure 6. As class C inherits from class B, the object ID B_1 is created right before the object ID C_1 on lines 6 and 7: as explained earlier, this is due to the way we initialize the objected attribute and the sequence of calls to constructors in an inheritance hierarchy in Java. The trace file contains the next method call, a creation of object of class A on lines 8 through 12 followed by the ID assigned to this new object on line 13. These calls are the results of execution of line 5 in Figure 5 namely “public A a = new A();”. The remainder of the trace file follows the execution of the example in similar fashion. Note that createMessage entries specify receive events as “nothing”, for example on line 3. This has to be corrected in a post processing step (see below). To make Trace.txt a valid XMI file, a header and a footer are added. To make the information regarding creation of

new object correct, any time there is more than one lifeline in a row, all lifelines except the last one are removed. We do not believe that there are instances, other than what is described here, in which more than one lifeline trace entry would appear in sequence. The last line is kept to obtain the created object's unique ID and this ID is placed into the preceding createMessage entry instead of the word "nothing". Lifelines are not needed past this point, so they are removed. These changes, performed on the trace file of Figure 15 are highlighted in the XMI file of Figure 16: the entire file is available in Appendix D.

```

1    <?xml version="1.0" encoding="UTF-8"?>
2    <trace:Log xmi:version="2.1"
      xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
      xmlns:trace="http://www.mdworkbench.com/trace"
      name="C:\Users\owner\workspace\Example3\Trace">
3    <messageLog bindToClass="example.C"
      messageSort="createMessage" signature="new example.C(int)">
4        <sendEvent covered="C_static"/>
5        <receiveEvent covered="C_1"/>
6        <sentFrom lineNumber="6" name="C.java"/>
7    </messageLog>
8
9
10   <messageLog bindToClass="example.A"
      messageSort="createMessage" signature="new example.A()">
11       <sendEvent covered="C_1"/>
12       <receiveEvent covered="A_1"/>
13       <sentFrom lineNumber="6" name="B.java"/>
14   </messageLog>
15
16   <messageLog bindToClass="example.A" messageSort="synchCall"
      signature="void example.A.simpleMultiplication(int)">
17       <sendEvent covered="A_1"/>
18       <receiveEvent covered="A_1"/>
19       <sentFrom lineNumber="16" name="A.java"/>
20   </messageLog>
21   <messageLog bindToClass="example.C" messageSort="synchCall"
      signature="void example.C.ifAndFor()">
22       <sendEvent covered="C_static"/>
23       <receiveEvent covered="C_1"/>
24       <sentFrom lineNumber="7" name="C.java"/>
25   </messageLog>
26   (...)
27   </trace:Log>

```

Figure 16 – Trace.xmi contents for the running example

To make `Trace.txt` a valid XMI file, a header and a footer are added. To make the information regarding creation of new objects correct, any time there is more than one lifeline in a row, all lifelines except the last one are removed. We do not believe that there are instances, other than what is described here, in which more than one lifeline trace entry would appear in sequence. The last line is kept to obtain the created object's unique ID and this ID is placed into the preceding `createMessage` entry instead of the word "nothing". Lifelines are not needed past this point, so they are removed. These changes, performed on the trace file of Figure 15 are highlighted in the XMI file of Figure 16: the entire file is available in Appendix D.

3.3 Control flow analysis

To instantiate the Control Flow Model of a source code, a parser determines classes, methods, control structures and calls to methods. The parser has been created with the JavaCC [17] tools, in particular JJTree, which enables automatic creation of syntax tree based parsers. An existing Java grammar JavaCC file for Java version 1.5, included in the JavaCC package, has been renamed to `cfparser.jjt` and modified to give us access to the string representations of sections of parsed code such as class name, method signatures, control flow structures and conditions. Recall that we only need to parse the abovementioned structures and therefore, in our context, we do not need all the productions of the complete Java grammar: for instance we do not need to parse every statement as we are not interested in generating bytecode or even binary code, we are not interested in performing complex source code analyses. Once the Java parser was created, we extended the `ControlFlowVisitor` class generated by the parser to generate (i.e., print information in a text file) control flow graphs. The parser does not support fully the Java 1.5 language, but merely the most common elements of the language that are important in our context. The parser can recognize class definitions, including inner class definitions, methods and method calls including method calls that are passed as parameters to other method calls and method calls that are inside condition statements. Most control flow structures can be handled by the parser including if, else if and else, while loop (but

currently not do-while), for loop (including for-each). The ?: construct is currently not supported, but doing so is not a technical challenge. Additionally, specifications of class attributes with method calls that do not appear at the beginning of the source code (i.e., appearing after a method specification in the source code), and therefore those method calls (e.g., signature, line number) are currently not recognized by the parser. Again, supporting such situations is not a technical challenge.

When a condition statement contains a method call, the method call will be represented in the control flow graph right ahead of the condition (outside of the conditional control flow construct). However, a method call that is passed as another method's call argument will be represented after, even though it will be executed before. Due to the nature of our analysis, this does not affect the accuracy of our results as the method calls are placed into the sequence diagram in the order they appear in the trace and the order of calls inside methods in the control flow graph is not consulted for that aspect (we only look at what method a method call belongs to). However, if in the future we intend to use the control flow graph to extract the order of method calls, the parser would need to be modified.

Figure 17 – Class B control flow graph for the running example

The output of the parser will be illustrated using the running example. Figure 17 shows the control flow diagrams for the methods in class B. The control flow diagrams for the remaining classes can be viewed in Appendix B. Class B has five methods, three of which contain a single line of code (i.e., `B()`, `callToA()`, and `getD()`), and one which contains none (i.e., `quickCall()`). The largest method, `ifAndFor` has condition nested inside a loop, nested inside a condition.

```

1  <controlflow:Class name="B">
2    <method signature="new B()">
3      <contains xsi:type="controlflow:MethodCall"
lineNumber="6" signature="new A()" isInMethod="new B()"
isInClass="B.java"/>
4    </method>
5    <method signature="example.B.B(int)">
6    </method>
7    <method signature="example.B.ifAndFor()">
8      <contains xsi:type="controlflow:Opt"
conditionDescription="attr==0">
9        <contains xsi:type="controlflow:MethodCall"
lineNumber="10" signature="new D()"
isInMethod="example.B.ifAndFor()" isInClass="B.java"/>
10       <contains xsi:type="controlflow:Loop"
conditionDescription="for (int i = 0; i_LT_totalLoops;
i)">
11         <contains xsi:type="controlflow:Alt"
conditionDescription="i==1">
12           <contains xsi:type="controlflow:MethodCall"
lineNumber="13" signature="getD(d)"
isInMethod="example.B.ifAndFor()"
isInClass="B.java"/>
13         </contains>
14         <contains xsi:type="controlflow:Alt"
conditionDescription="else i==1">
15           <contains xsi:type="controlflow:MethodCall"
lineNumber="16" signature="callToA()"
isInMethod="example.B.ifAndFor()"
isInClass="B.java"/>
16         </contains>
17       </contains>
18     </contains>
19   </method>
20   <method signature="example.B.getD(D)">
21     <contains xsi:type="controlflow:MethodCall"
lineNumber="21" signature="get()"
isInMethod="example.B.getD(D)" isInClass="B.java"/>
22   </method>
23   <method signature="example.B.callToA()">
24     <contains xsi:type="controlflow:MethodCall"
lineNumber="22" signature="threeCalls(this)"
isInMethod="example.B.callToA()" isInClass="B.java"/>
25   </method>
26   <method signature="example.B.quickCall()">
27   </method>
28 </controlflow:Class>

```

Figure 18 – Class B from the running example displayed in control flow graph

The actual text file representing the control flow graph of class B is shown in Figure 18. All method calls are shown inside their respective methods, while the method calls that

are inside conditional sections are shown nested into their respective structures. Lines 2 through 4 detail the class constructor. Note that class B has two constructor methods specified in the model, one has signature `new B()` (this is different from what the default constructor, i.e., the default constructor's signature, if it was explicitly defined in the code, would be `example.B.B()`) and the other is explicitly specified `example.B.B(int)`. In Figure 18 the parser recognises that an attribute `a` of type A in class B will be initialized during constructor's execution (see Figure 5) so a call to `new A()` is included in the parser output on line 3. The explicit constructor (which takes an `int` parameter) appears on lines 5 and 6. We chose to represent the two constructors separately because they are physically separate in the source code and the parser would have to handle extra complexity if we were to insert the method calls from attribute initialization section into each constructor of the class. Instead of joining the attribute initialization with the class constructors in the parser, we will handle it in the model transformation step.

Notice that not all the details of the source code appear in the control flow diagram: for instance, assignment statements do not appear; since we are only interested in class and method definitions, method calls, and control flow structures.

Lines 7 through 19 describe the `ifAndFor` method. Lines 8 through 18 represent everything that is inside the `if(attr==0)` condition including a method call, line 9, and a for loop, lines 10 through 17. The if-then-else construct is represented as two separate conditions one after the other, lines 11 through 13 and 14 through 16. Each of these sections in turn contains their respective method call.

4 TRACE AND CONTROL FLOW MODELS

To formalize our transformation from raw data in the form of execution trace and SUS source code (control flow graph) into UML 2.1 sequence diagrams, we devised two models: a trace model and a control flow model. Through a transformation procedure described in section 5, output data is returned in the UML 2.1 format [UML Superstructure]. The trace model is discussed in section 4.1 and the control flow model is discussed in section 4.2.

4.1 Trace model

The purpose of the trace model is to present execution trace data in a format that is readable by transformation algorithms described in section 5. It holds records for method calls, referred to as messages in this thesis, made during execution. The trace model is described in a XML Schema document (Appendix C). An example of a model file and a procedure to obtain it has been discussed in section 3.2.5.

4.1.1 Model elements

The trace model (Figure 19) is designed to be very close in structure to UML 2.1 Superstructure's Message components in order to make transformations as simple as possible. In particular, the trace model's elements Log, MessageLog, MessageLogOccurrenceSpecification and MessageSort map to the UML's Interaction, Message, MessageOccurrenceSpecification and MessageSort respectively.

Figure 19 – Trace model

Log represents a single program execution and contains MessageLogs. A MessageLog stands for a message sent to signal the start of an execution. A MessageLog has two MessageLogOccurrenceSpecifications: One connects the message to its sending object and the other to the receiving object through the `receivedEvent` and `sendEvent` rolenames. Covered attribute is a String containing the ID of an object, to be eventually translated into a lifeline. MessageLog's attributes include signature, messageSort and bindToClass. MessageSort specifies whether the MessageLog is representing a constructor or a synchronous message. A UML MessageLog's messageSort attribute is going to map to the UML's Message's messageSort which can be a "synchronous call", "asynchronous call", "asynchronous signal", "create message", "delete message" and "reply". Due to the scope of this work, no other message sort than constructor or synchronous message is expected in our trace. Signature contains a String with the message's signature, which is in the format "returntype package.class.calledmethodname(arguments)". BindToClass contains the name of the class of which the object is an instance. In the case of a static call, bindToClass will contain the class defining this static method. This information allows the transformation algorithm to determine the specific class and method invoked by the method call as method call can bind to more than one method during execution.

`MessageLog` contains a `SourceLocation` which specifies the location (string name of the class and the integer `lineNumber` at which the call is made) in the source code from where the method call connected to the message has been sent. This, along with the attribute `bindToClass` will allow us to take a `MessageLog` from the trace and match it to correct elements of the control flow model instance obtained from the source code static analysis (i.e., the right method in the callee class).

4.1.2 Example instance of the model

Each instance of the trace model is built as described in section 3.2.

Figure 20 – Partial representation of the running example trace as trace model instance

Figure 20 is a visual representation of the information already presented in Figure 16. Each MessageLog instance represents one trace entry. The first MessageLog in the

diagram represents the first method call made in the running example execution, `new example.C()` method call. The “createMessage” method call was dynamically bound to method with signature `new example.C()` in class `C.java`. This `MessageLog` has two `MessageLogOccurrenceSpecifications`: one to represent from where the method call was made and the other to represent the receiving object. `new example.C()` was sent from the main method of class `C` so we use the term “`C_static`” in associated `MessageLogOccurrenceSpecification` to denote this origin. This method created a new object `C_1` represented by the other `MessageLogOccurrenceSpecification`. The remaining method calls are likewise represented in the trace model instance in sequential order in which they occurred during execution.

4.2 Control flow model

The role of the control flow model is to capture code structure in terms of method calls performed by methods. One particularly important aspect of the control flow model is the design of nested method calls and control flow structures into other control flow structures and methods. The control flow model is described in an XML Schema document found in Appendix C

4.2.1 Model elements

The control flow model, shown in Figure 21, is designed to allow us to accurately locate method calls from the source code based on matching `MessageLogs` from the trace model and then place them into the UML sequence diagram structure. Knowledge of a method call’s host method and, if the method call is inside a condition, its control flow structures will allow us to accurately construct the sequence diagram of execution with minimal dynamic (trace) data.

The control flow model contains each `Class` that appears in the analyzed source code. A `Class` may contain `Methods` which in turn contain `MethodCalls` and `ConditionalSections`, specializations of `CodeSections`. `Method` represents both

regular methods and constructors. A ConditionalSection can have nested CodeSections. A ConditionalSection can be specialized as an Opt, Alt or Loop. A Loop has a LoopType set as either “for”, “do” or “while”. Each ConditionalSection has a string attribute conditionDescription that contains the actual condition. MethodCall keeps track of the lineNumber from where it lies in the method body. MethodCall’s attribute isInMethod contains the signature of the method this method call is in. For MethodCalls that are not inside a ConditionalSection, method.signature (navigating rolename method and accessing attribute signature of the calling method) will be the same as isInMethod. However, MethodCalls that are contained inside a ConditionalSection do not have direct access to the method rolename and therefore need to carry the isInMethod attribute. Attribute isInClass contains the name of the Class the MethodCall is in, and is needed for similar reasons as isInMethod.: for instance in case of a direct call, isInMethod equals method.methodClass.name.

Figure 21 – Control flow model

A `ConditionalSection` specialized as `Alt` does not contain information about true/false branches because they are not handled as such in this work. The distinction between `Opt` and `Alt` is used in the mapping algorithm described in Chapter 5 hence they are kept as separate entities here.

4.2.2 Example instance of the control flow model

Each instance of the control flow model is built by the source code parser described in section 3.3. While the loop condition expression is gathered by the parser, currently `Loop` attributes `min` and `max`, shown in Figure 21, are not initialized. To initialize these attributes, we would need several execution traces and then need to recognize similar loop executions in several traces, i.e., transform scenarios diagrams into sequence diagrams, which is out of the scope of this thesis.

Figure 22 – Partial representation of the control flow model instance for the running example

Figure 22 is a visual representation of the information already presented in Figure 17 and Figure 18: Class B. Class B is shown with its four methods. Method calls and control flow structures are shown in their respective methods. Note that the method calls inside conditional sections do not have direct access to the method they belong to hence the need for `isInMethod` and `isInClass` attributes as discussed previously.

4.3 The UML Superstructure (excerpt)

In this section we describe part of the UML superstructure [18], specifically the part of the superstructure specifying interactions (i.e., sequence diagrams), which is relevant to

the transformations we discuss next. As UML Superstructure is large, we will focus only on elements used in this work.

Figure 23 – Messages in UML Superstructure [18]

To construct a UML sequence diagram of the SUS execution, we use two Messages, show in Figure 23, to represent a MethodCall. The first Message represents the call being made and the second represents the reply (or end of execution). Each Message has two MessageEnds instantiated as MessageOccurrenceSpecifications, one representing sending and the other receiving of the Message. Their role names (from

Message) are `sendEvent` and `receiveEvent` respectively. All Messages in a sequence diagram are owned by an `Interaction` and their attributes include `name` and `MessageSort`. In this work, we store the signature of the method being called as `name` and `MessageSort` can be set to “`createMessage`” or “`synchCall`” for Message representing the call being made and “`reply`” for Message returning from the call. Each Message associates to a `Connector` which specifies communication link between two Lifelines. A `Connector` has two `ConnectorEnds`, one to each Lifeline.

Figure 24 – Lifelines in UML Superstructure [18]

Lifelines are used to represent individual participants in the Interaction, for example objects. As seen in Figure 24, the Lifeline represents a `ConnectableElement`. In this work the `ConnectableElement` is instantiated as

Property. Lifeline belongs to the Interaction and is covered by, amongst others, OccurrenceSpecification (instantiated as MessageOccurrenceSpecification in this work) and InteractionFragment (instantiated as CombinedFragment here). OccurrenceSpecifications are ordered along the Lifeline. This order specifies the sequence in which events happen. In our case MessageOccurrenceSpecifications are ordered in the sequence their respective Messages are sent and received.

Figure 25 – Occurrence and Execution Specifications in UML Superstructure [18]

We use *ExecutionSpecification* instantiated as *BehaviourExecutionSpecification* to represent execution of a method.

BehaviorExecutionSpecification is delineated by two MessageOccurrenceSpecifications through rolenames start and finish. The MessageOccurrenceSpecification denoted by start is the receiveEvent of the Message that represents method call opening the method execution. Subsequently, the MessageOccurrenceSpecification denoted by finish is the sendEvent of the reply Message for the same method call representation. MessageOccurrenceSpecifications are instantiated from InteractionFragments and are owned by Interaction as fragments.

Figure 26 – Events in UML Superstructure [18]

OccurrenceSpecification, as per Figure 25, points to an Event. Event, Figure 26, can be instantiated in our work as CreationEvent, SendOperationEvent and ReceiveOperationEvent. We use Operations to represent methods. They are owned by Classes which we use to represent the classes of the SUS.

Figure 27 – CombinedFragment in UML Superstructure [18]

We use CombinedFragment construct, shown in Figure 27, to represent control flow structures. It is an InteractionFragment owning an InteractionOperand that specifies guard (InteractionConstraint). The guard is used to specify condition statement of the condition or loop that the CombinedFragemnt represents. InteractionOperand owns fragments representing elements contained inside the condition or loop. In this work these would be either MessageOccurrenceSpecifications or nested CombinedFragments. CombinedFragment has an attribute interactionOperator which can be set to any value enumerated by InteractionOperatorKind but here we use “opt”, “alt” and “loop”.

5 MAPPING THE MODELS TO THE SCENARIO DIAGRAM

To arrive at the complete Scenario Diagram representation of an execution, four steps will be taken. The first step, described in section 5.1, is to link the traced executions of method calls (`MessageLogs` in the trace model) to method calls in the source code (`MethodCalls` in the control flow model). In the next step, found in section 5.2, sequence diagram elements unique to this method execution are created and added to the model while elements that the method execution might share with previous method executions are either located elsewhere in the sequence diagram (if already created) or, if they do not yet exist, created and added. The third step, detailed in section 5.3, is done in cases where the method call was found to be inside of a conditional section which is then modeled in the sequence diagram model as a Combined Fragment. The final step, section 5.4, contains an explanation of how our approach determines when all method calls from a particular method have been executed and a reply Message corresponding to the method's opening method call Message can be inserted into the sequence diagram. In this document we use the Object Constraint Language (OCL) to describe the model transformations. The implementation of the transformations described here was done using MD Workbench, a powerful Eclipse based IDE for code generation and Model transformation [19]: section 5.5.

Note that the definition of the mapping between information in the trace model instance and control flow model instance has been set to be the responsibility of a third party class, which we named `Matching`. This allows us to formally specify the mapping without having to modify any of the three models (which are considered as data only): trace model, control flow model, and more so the UML model.

5.1 Linking trace execution logs to source code

For every MessageLog entry into the trace model, there exists a MethodCall in the control flow model.

```
Matching :: matchMessageLogToMethodCall(MessageLog ml) : MethodCall
post : result = MethodCall->allInstances->select(mc |
    ml.signature = mc.signature
    and ml.sentFrom.name = mc.isInClass
    and ml.sentFrom.lineNumber = mc.lineNumber )
```

Figure 28 – Matching MethodCall:ControlFlow to MethodLog:Trace

Class MethodCall in the control flow model is located by matching a MessageLog's signature, sentFrom class name and sentFrom line number as shown in Figure 28. When matching signatures, it is not the full signature string that is compared but only the method name as the two models' signatures are not in the same format. Specifically, a typical method signature (string) in the control flow model looks like "method(argument)" whereas a typical method signature (string) in the trace model looks like "type packet.class.method(argumenttype)". For example threeCalls(this) in the control flow model matches to int example.A.threeCalls(B) in the trace model.

5.2 Basic sequence diagram elements

Every UML model created by our transformation contains an encompassing element Model which in turn has packagedElement Collaboration. Interaction is an ownedBehaviour of Collaboration. Our algorithm will create these and then iterate through MessageLog elements of the trace model to create the full sequence diagram corresponding to our SUS trace and source code.

For every MessageLog entry in the trace model, there are elements in the sequence diagram that are specific to this MessageLog and are not shared with another MessageLog's elements. If MessageLog is of sort createMessage, these unique elements are one Message, two MessageOccurrenceSpecifications, a SendOperationEvent and a CreationEvent. If the MessageLog sort is synchCall

then these elements are two Messages (one for the call and one for the return), four MessageOccurrenceSpecifications (two for the call message and two for the return message), two SendOperationEvents, two ReceiveOperationEvents and a BehaviourExecutionSpecification. In this work, we will not use createMessage type, but we allow for its usage in future work. Other elements related to the MessageLog may be shared with other MessageLogs and are created for a MessageLog only if the matching ones do not yet exist in the model. These elements include:

- one or two Lifelines, and Properties representing these Lifelines,
- an Actor for the initial Lifeline corresponding to the first MessageLog in the trace
- Classes for the remaining Lifelines
- Connector
- two ConnectorEnds
- Operation as well as elements representing control flow, namely:
CombinedFragment, InteractionOperand, InteractionConstraint,
and OpaqueExpression.

Comments may be attached to sequence diagram elements. They are not necessary for the sequence diagram to be complete, but the algorithm presented here adds Comments to Messages, Operations and CombinedFragments, indicating their relationship to the source code such as class name and line number where the related method calls are found.

```
Matching :: SDElementsForMessageLog(MessageLog ml)
post : Interaction.allInstances->exists(i |
    i.message->exists(m_s : Message |
        Matching.mapSendMessage(m_s, ml))
    and
    i.message->exists(m_r : Message |
        Matching.mapReplyMessage(m_r, ml))
    and
    i.fragment->
        select(oclIsKindOf(BehaviourExecutionSpecification)
            ->exists(bes| Matching.mapToBES(bes, ml)))
```

Figure 29 – Creating unique sequence diagram elements for MessageLogs

Figure 29 outlines in OCL the abovementioned specification for one MessageLog at a time: this would have to be done for each message log of a log. The result is the existence of sequence diagram elements corresponding to each MessageLog. For each MessageLog, the matching relies on other operations of class Matching, namely mapSendMessage(), mapReplyMessage(), and mapToBES() which are outlined in Figure 30, Figure 36 and Figure 37, respectively, and are specific to what needs to be matched.

```

Matching :: mapSendMessage(m: Message, ml : MessageLog)
post : m.name = ml.signature
    and
    m.ownedComment.body = matchMessageLogToMethodCall(ml).isInClass
        + "{"
        + matchMessageLogToMethodCall(ml).lineNumber
        + "}"
    and
    if ml.messageSort = MessageSort::createMessage
    then m.messageSort = MessageSort::createMessage
    else m.messageSort = MessageSort::synchCall
    endif
    and
    m.connector.end->exists(end : ConnectorEnd |
        Lifeline.allInstances->exists(ll |
            Matching.mapSendLifeline(ll, ml)
            and
            end.role.type = ll.represents) )
    and
    m.connector.end->exists(end : ConnectorEnd |
        Lifeline.allInstances->exists(ll |
            Matching.mapReceiveLifeline(ll, ml)
            and
            end.role.type = ll.represents) )
    and
    Matching.mapSendLifeline(m.sendEvent.covered, ml)
    and
    Matching.mapReceiveLifeline(m.receiveEvent.covered, ml)

```

Figure 30 – Mapping sending Message for MessageLog

Figure 30 shows the mapping between a message log (Trace) and a message (UML). A Message contains information (name, comment and messageSort) matching a MessageLog and is associated with ConnectorEnds and Lifelines. MessageOccurrenceSpecification elements associated with this Message, specifically the send event (m.sendEvent) and the receive event (m.receiveEvent),

are also matched at this time to their respective lifelines. The ConnectorEnds connect the Message to its sending and receiving Lifelines and are also matched to the message log characteristics, specifically the sender and receiver objects. Figure 31 and Figure 32 specify the mapping between lifelines and message log characteristics.

```
Matching :: mapSendLifeline(l1: LifeLine, ml: MessageLog)
post : l1.name = ml.sendEvent.covered
      and
      (Matching.mapSendClass(l1.represents.type, ml)
       or
       Matching.mapSendActor(l1.represents.type, ml)
      )
      and
      l1.represents.name = ml.sendEvent.covered
      and
      l1.owner.owner.ownedAttribute -> exists (l1.represents)
```

Figure 31 – Mapping sending Lifeline for MessageLog

```
Matching :: mapReceiveLifeline(l1: LifeLine, ml: MessageLog)
post : l1.name = ml.receiveEvent.covered
      and
      l1.represents.name = ml.receiveEvent.covered
      and
      Matching.mapReceiveClass(l1.represents.type, ml)
      and
      l1.owner.owner.ownedAttribute -> exists (l1.represents)
```

Figure 32 – Mapping receiving Lifeline for MessageLog

Lifeline and Property names are derived from MessageLog's sendEvent.covered or receiveEvent.covered data, which represent objects whose behaviour has been traced. As Lifelines may be used by more than one Message, the constraints on Lifeline mapping do not need a precondition since they may or may not exist prior to the processing of a MessageLog. Note that Lifeline represents a Property. Lifeline and Property will either be located by the algorithm or created if they do not already exist. A Lifeline will already exist if it has already been in use by a Message created from previous MessageLog. A Lifeline is owned by an Interaction which is owned by a Collaboration. A Collaboration owns all Properties so any time a Property is created, it needs to be added to the

Collaboration. Furthermore, a Property may be shared amongst many Interactions whereas a Lifeline exists only inside the context of an Interaction.

```

Matching :: mapReceiveClass(c: Class, ml: MessageLog)
post : Interaction.allInstances->exists(i |
    i.owner.owner.packagedElement->includes(c)
    and
    c.name = ml.bindToClass
    and
    c.ownedOperation->exists(operation |
        operation.name = ml.signature
        and
        operation.ownedComment.body =
            matchMesseageLogToMethodCall(ml).isInClass + " {" +
            matchMesseageLogToMethodCall(ml).lineNumber.toString()
            + "}"
    )
)

```

Figure 33 – Mapping receiving Class for MessageLog

In Figure 32, mapReceiveLifeline references a class constrained by mapReceiveClass shown in Figure 33. The class is owned by Model and its name is derived from ml.bindToClass.

The equivalent line in Figure 31 gives an option between Lifeline's role name represent.type mapping to Class or Actor through mapSendClass() and mapSendActor() respectively. This is because the algorithm presented here does not create the Class for sending Lifeline. The reason is that the scope of this algorithm is limited to single threaded execution, and therefore only the first MessageLog in the log will not already have its sending Lifeline, Property and Class created previously (Figure 34). When creating the first Message in the Model, or a Message from an uninstrumented class, the sending Class does not exist yet so instead of Class, Actor will be created at the same time as Lifeline and Property as seen in Figure 35.

```

Matching :: mapSendClass(c: Class, ml: MessageLog)
pre : Interaction.allInstances->exists(i |
    i.owner.owner.packagedElement
    -> exists (c)
    and
    c.name = matchMessageLogToMethodCall(MessageLog
ml).isInClass
    and
    i.lifeline.allInstances->exists(ll |
    ll.represents.type = c
    and
    ll.represents.name = ml.sendEvent.covered)
)
)

```

Figure 34 – Mapping sending Class for MessageLog

Figure 34 describes the case where associated send Lifeline, and therefore Property and Class already exist prior to processing of the current MessageLog.

```

Matching :: mapSendActor(a: Actor, ml: MessageLog)
pre : not Interaction.allInstances->exists(i |
    i.lifeline->exists(ll |
    ll.represents.name = ml.sendEvent.covered)
)
)
post : Interaction.allInstances->exists(i |
    i.owner.owner.packagedElement->includes(a)
    and
    a.name = matchMessageLogToMethodCall(MessageLog
ml).isInClass
    and
    i.lifeline->exists(ll |
    ll.represents.type = a
    and
    ll.represents.name = ml.sendEvent.covered)
    and
    i.owner.ownedAttribute->exists(p | ll.represents = p)
)
)

```

Figure 35 – Mapping sending Actor for MessageLog

Figure 35 describes the case where Actor is created and associated to a new Lifeline and Property. This happens when the sending Lifeline does not already exist in the Model prior to the current MessageLog being processed.

Class owns operation whose name matches MessageLog signature and ownedComment contains source code coordinates.

```

Matching :: mapReplyMessage(m: Message, ml : MessageLog)
pre  : not Interaction.allInstances->exists(i | i.message->includes(m))
post : let llRec: LifeLine = LifeLine.allInstances->select(ll |
      Matching.mapReceiveLifeLine(ll,ml))
      let llSend: LifeLine = LifeLine.allInstances->select(ll |
      Matching.mapSendLifeLine(ll,ml))
      in
      Interaction.allInstances->exists(i | i.message->includes(m)
      and
      m.name = ml.signature
      and
      m.ownedComment =
        Matching.matchMessageLogToMethodCall(ml).isInClass + "{" +
        Matching.matchMessageLogToMethodCall(ml).lineNumber + "}"
      and
      m.messageSort = MessageSort::reply
      and
      m.connector.end->includes(ConnectorEnd.allInstances
      ->select(ce| ce = llRec.represents))
      and
      m.connector.end->includes(ConnectorEnd.allInstances
      ->select(ce| ce = llRec.represents))
      and
      mapReceiveLifeline(m.sendEvent.covered, ml)
      and
      mapSendLifeline(m.receiveEvent.covered, ml)
      )

```

Figure 36 – Mapping reply Message for MessageLog

Reply Message, shown in Figure 36, is very similar to the sending Message except for its type, which is by default “reply”, and the lifelines are reversed. The lifeline sending the reply Message is the lifeline that had received the sending Message and the lifeline receiving the reply Message is the lifeline that has sent the sending Message.

```

Matching :: mapBES(bes BehaviourExecutiunSpecifcation, ml: MessageLog)
pre  : not BehaviourExecutionSpecifcation.allInstances->includes(bes)
post : let mSend: Message = Message.allInstances->select(m |
      Matching.mapSendMessage(m,ml))
      let mRep: Message = Message.allInstances->select(m |
      Matching.mapReplyMessage(m,ml))
      in
      BehaviourExecutionSpecifcation.allInstances->includes(bes)
      and
      bes.name = ml.signature
      and
      bes.start = mSend.receiveEvent
      and
      bes.finish = mRep.sendEvent

```

Figure 37 – Mapping BehaviourExecutionSpecifcation for MessageLog

A BehaviourExecutionSpecification, as specified in Figure 37, is created for each “synch” MessageLog and it references the sending Message’s receiving MessageOccurrenceSpecification as its start and replying Message’s sending MessageOccurrenceSpecification as its finish.

5.3 Combined Fragments

Combined fragments represent control flow structures that are enclosing messages (i.e., method calls) associated with log entries in the SUS execution trace. If a methodCall is inside a conditionalSection, then the corresponding Message in the sequence diagram will be inside a CombinedFragment. Furthermore, if such a conditionalSection is nested inside another conditionalSection, then the resulting CombinedFragment will be inside another CombinedFragment representing nesting conditionalSections, and so on.

More than one MessageLog can be associated with a CombinedFragment, i.e., a conditional section in the code can contain several method calls, so a new CombinedFragment will only be created once the algorithm determines that an existing element (i.e., combined fragment) with appropriate settings does not already exist in the Model.

Calls made from a loop may execute widely different paths and be difficult to represent graphically. Because each loop in the automatically generated diagram represents a single transversal of the loop, there will never be a case where if and else sections are found inside a single “alt” CombinedFragment as they are in the manually generated diagram. Our approach does not recognize repeated iterations of a loop as one CombinedFragment. The problem of combining several scenario diagrams into one sequence diagram, thereby merging alternative and repeated executions (sub-traces) is deferred to future work. Instead, a new CombinedFragment will be added to the sequence diagram for each iteration of a loop. The algorithm contains a section that recognizes whether the two Messages, both inside the loop, should be placed inside the

same CombinedFragment as they may or may not be elements of the same iteration. The algorithm does this by analyzing the line number of the method calls. If the two consecutive method calls are made from the same .java file and the earlier method call has a higher line number than the latter method call or if they have the same line number and the same signature, the algorithm assumes that the loop has been traversed again and that the two method calls are not from the same iteration.

If the latter method call is inside an “else” section, the previous method call is checked to see if its condition name indicates that it belongs to the corresponding “if” section. If it does, the two representations of the method call are placed inside separate CombinedFragments representing the same loop.

For the purpose of this work, Opt and Alt constructs from the control flow graph are treated in the same manner (Alt will not contain two options as it would in a sequence diagram) except that the terms “Opt” and “Alt” carry into the final diagram. Additionally an Alt construct has slightly different handling in the algorithm presented in Figure 38,

```

1  if (mc.containingSection != null) {
2      breakLoop = 0;
3      lastMessageLineNumber = interaction.message.last().lineNumber();
4      lastMessageClass = interaction.message.last().class();
5      lastMessageIsInMethod = interaction.message.last().method();
6      if (mc.isInClass.equals(lastMessageClass)
          && mc.isInMethod.equals(lastMessageIsInMethod)
          && ((mc.lineNumber < lastMessageLineNumber)
              || ((mc.lineNumber == lastMessageLineNumber)
                  && ml.signature.equals(
                      interaction.message.last().signature.toString())))) {
7          breakLoop = 1;
8      }
9      containingSection = mc.containingSection;
10     while (containingSection != null) {
11         bFound = false;
12         lookingforfragment = interaction.message.last().
            sendEvent.enclosingOperand.owner;
13         if (lookingforfragment.operand.first().fragment.last().
            getClass().toString().equals("class com.sodius.mdw.
            metamodel.uml21.internal.impl.
            BehaviorExecutionSpecificationImpl")){
14             lookingforfragment = null;
15         }
16         while (lookingforfragment != null){
17             if (lookingforfragment.interactionOperator == ALT LITERAL

```

```

        && ((lookingforfragment.name.contains(
        containingSection.conditionDescription)
        && lookingforfragment.name.contains("else"))
        || (containingSection.conditionDescription.contains(
        lookingforfragment.name)
        && containingSection.conditionDescription.contains(
        "else")))
        && !containingSection.conditionDescription.equals(
        lookingforfragment.name)
        && mc.isInClass.equals(lastMessageClass)
        && mc.isInMethod.equals(lastMessageIsInMethod)) {
18     breakLoop = 1;
19 }
20 else if (containingSection.getClass == "Loop"){
21     breakLoop = 0;
22 }
23 else if ((lookingforfragment.name ==
        containingSection.conditionDescription)) {
24     if (breakLoop == 0){
25         bFound = true;
26         combFragment = lookingforfragment;
27         if (containingSection == mc.containingSection){
28             innermostCombFragment = lookingforfragment;
29         }
30     }
31     break;
32 }
33     lookingforfragment =
        lookingforfragment.enclosingOperand.owner;
34 }
35 if (bFound == false) {
36     createCombinedFragment();
37 }
38 if ((fragmentqueue.fragment.last() != null)){
39     innerfrag = fragmentqueue.fragment.last();
40     innerfrag.enclosingOperand = combFragment.operand.first();
41 }
42 if (bFound == false) {
43     fragmentqueue.fragment.add(combFragment);
44 }
45 }
46     containingSection = containingSection.containingSection;
47 }
48 if ((fragmentqueue.fragment.last() != null) ){
49     enclosingInteraction.operand.first().fragment.add(
        fragmentqueue.fragment.last());
50 }
51     enclosingInteraction = innermostCombFragment;
52 }

```

Figure 38 – Pseudo code for nesting CombinedFragments

Figure 38 contains pseudo code for arranging CombinedFragments. Lines 2 through 49 are executed only if the MethodCall `mc` is inside a condition. On line 2 variable `breakLoop` is initialized. This variable will be used to flag when a loop needs to be interrupted for the next iteration. Variable `mc` refers to the MethodCall associated with the currently processed MessageLog. Because a new CombinedFragment will only be created if a matching CombinedFragment does not already encompass a previous Message in the diagram, lines 3, 4 and 5 are used to obtain basic information about the last Message. The condition on line 6 checks whether the current Message is in the same loop iteration as the last Message. If the current Message is in the same method and class as the last one, but ahead of the last one with respect to line numbers, variable `breakLoop` will be set on line 7. The `breakLoop` variable can also be set to 1 if one of the two Messages is inside of an “if” condition and the other is inside of the matching “else” (lines 14 through 16). The `containingSection` variable is set on line 9 and the while loop that will iterate through all nested containingSections containing `mc` starts on line 10. The `lookingforfragment` variable refers to the CombinedFragment containing the last Message. `lookingforfragment` is only set if the last fragment in the CombinedFragment is not a BehaviourExecutionSpecification. If the last fragment is a BehaviourExecutionSpecification, the condition surrounding the last Message is not from the currently executing method. The while loop starting on line 13 will iterate through CombinedFragments surrounding the last Message, in search for a CombinedFragment that should also contain the current Message (condition check on line 23). Even if fragments match, the two Messages will only be placed in the same CombinedFragment if `breakLoop` variable equals 0. On each traversal of “loop” containingSection, the `breakLoop` variable is reinitialized to 0 so the next outward nesting CombinedFragment may contain both Messages if other required conditions are true. If a condition around current Message does not exist around the previous Message, or they are in separate iterations of a loop, a new CombinedFragment is created for the current Message (line 36). Whether we have created a number of new

nesting CombinedFragmetns or matched them to existing ones, the innermost fragment must be known (line 28 or hidden inside the function on line 36). Newly created CombinedFragmetns are temporarily stored in the fragmentqueue queue (line 43). On subsequent iterations of the containingSection loop, fragment existing in the queue will be enclosed in newly matched or created CombinedFragment (lines 39 and 40). Once the containingSection loop has completed, if there are any fragments in the fragmentqueue, they are inserted in the enclosingInteraction (Interaction or InteractionFragment that owns the BehaviourExecutionSpecification from which the current Message is sent) on line 49. The enclosingInteraction is going to be the variable which will be set as the InteractionFragment owning the current Message so the innermostFragment is set as the new enclosingInteraction on line 51.

5.4 Ordering of send and reply Messages

In a sequence diagram produced by our approach, Messages are ordered from top to bottom in chronological order of occurrence. The ordering of send Messages is simple enough; they are ordered in the order found in the trace. Reply Messages corresponding to the completion of execution of the method are not collected in a trace during execution so their ordering with respect to other send and reply Messages has to be determined by the algorithm.

When a MessageLog is processed and its sequence diagram elements are added to the UML Model, the reply Message is placed into a last-in-first-out (LIFO) queue outside of the Model. This Message is only taken out of the queue and reinserted into the Model once the algorithm determines that the method invoked by the associated send Message has completed. The algorithm keeps track of the last reply Message in the LIFO and it knows, from the trace model data, the name of the method containing the method call associated with the currently processing MessageLog. As long as the signature of the last reply Message in the LIFO queue and the method housing the last processed

MessageLog match, the BehaviourExecutionSpecification in the sequence diagram associated with this method should not be closed.

During the processing of every MessageLog, if the LIFO queue is not empty, a check is made to verify whether the reply Message' sendEvent operation name matches the current MethodCall's isInMethod. If they match, nothing is removed from the queue.

The match indicated that the current MessageLog corresponds to a Message that is inside the method invoked by the send Message connected to the reply Message on top of the queue. If the method names do not match, then it is understood that the method has completed execution and the reply Message can be inserted into the sequence diagram. When all MessageLogs have been processed, the reply Messages remaining in the queue are moved to the Model.

With constructors, unlike with regular methods, we cannot compare signatures to determine that one method was called by another. In this case, we compare the receive Lifeline of the last Message in the reply queue and the send Lifeline of the current Message. If their Lifelines match and we confirm that the method invoked by the last Message in the reply queue is a constructor by verifying that it starts with "new " then we know that current method call is contained inside the constructor execution. The last Message in the reply queue is not removed from the queue.

5.5 Transformation with MDWorkbench.

MDWorkbench is an Eclipse-based IDE for model driven development. This tool provides a model transformation capability and supports widely used concepts such as Ecore, UML and XML Schema. Transformations are rule-based and can transform any number of source models into any number of target models, or may modify the source model only. Model transformations are specified by writing a set of rules in the MQL language. [19]

In this work we specify the models using XML Schema (Appendix C). Instances of the trace and control flow models (XMI input files) are manipulated using rules in a rule set, producing an instance of the UML sequence diagram model (XMI output file). This XMI output file is ready to be used by any UML CASE tool that supports the UML model. To visualize the results of the transformation, we use RSA [1].

```

1      public ruleset reverseengineering(in sourceT : trace, in sourceCF
      : controflow, out target : uml21) {
2
3          public rule main() {
4              // create model and collaboration
5              model = target.create("Model");
6              model.name = "Reverse Engineering Model";
7              collaboration = target.create("Collaboration");
8              collaboration.name = "Reverse Engineering
              Collaboration";
9              model.packageElement.add(collaboration);
10             foreach (log : trace.Log in
                  sourceT.getInstances("Log")) {
11                 interaction = target.create("Interaction");
12                 @createInteraction(log, interaction,
                  messagequeue, fragmentqueue, model,
                  collaboration);
13                 collaboration.ownedBehavior.add(interaction);
14             }
15         }
16     }
17 }

```

Figure 39 – Transformation ruleset (excerpt about Model, Collaboration and Interaction)

Figure 39 displays a segment of the ruleset file used in this work. On line 1, the input (trace and controflow) and output (uml21) models are specified. To begin, Model and Collaboration elements of the output target are created and packaged appropriately according to the UML specification (lines 5 through 9): a model is created for each trace, and this model contains one collaboration (i.e., sequence diagram). The script then transverses the Logs of the input trace model (line 10). For each Log, an Interaction will be created (line 11) and stored (line 13) in the Collaboration. On line 12 a call is made to method `createInteraction` (not shown in the Figure, but can

be seen in the complete transformation source code in Appendix E) which will create the remaining elements of the sequence diagrams given the two input models.

```

1      bFound = false;
2      foreach (itr : uml21.Lifeline in target.getInstances("Lifeline"))
3          {
4              if (itr.represents.name == ml.sendEvent.covered){
5                  sendProperty = itr.represents;
6                  sendLifeline = itr;
7                  bFound = true;
8                  break;
9          }
10
11      if (bFound == false){
12          actor = target.create("Actor");
13          actor.name = mc.isInClass;
14          model.packageElement.add(actor);
15
16          sendProperty = target.create("Property");
17          sendProperty.name = ml.sendEvent.covered;
18          sendProperty.type = actor;
19
20          collaboration.ownedAttribute.add(sendProperty);
21
22          sendLifeline = target.create("Lifeline");
23          sendLifeline.name = ml.sendEvent.covered;
24          sendLifeline.represents = sendProperty;
25          interaction.lifeline.add(sendLifeline);
26      }

```

Figure 40 - Transformation ruleset (excerpt about Lifelines)

Figure 40 displays an excerpt of implementation of the OCL specifications related to the Lifelines as shown in Figure 31. Lines 1 through 9 search through all existing Lifelines in the Model for a Lifeline whose name matches the object ID of the sending object denoted by `ml.sendEvent.covered`. As specified in Figure 31, sending Lifeline can be representing a Property whose type is a Class if the Lifeline already exists in the Model or an Actor if this is the first time the Lifeline appears as we process MessageLogs. If the condition on line 3 is true, then the Lifeline already exists and the already existing Property it represents is connected to an already existing Class through role name type as specified in Figure 34. If the matching Lifeline is not found then the condition on line 10 is true and Actor, Property and Lifeline are crated as specified by the OCL expression in Figure 35.

5.6 Example

To briefly illustrate the transformation, Figure 41 through Figure 45 display the scenario diagram obtained by executing the Example program with parameter “4” passed to it. Passing “4” will cause the execution to pass through the for loop four times.

Figure 41 – Scenario diagram for the running example, part 1

Figure 41 shows the first three method calls occurring in the example. From the main method a call to `new C(int)` is made. Class C inherits from B which has an attribute A created at initialization. A’s constructor contains a call to `simpleMultiplication(int)`. Note that the return Message text may be removed and the send Message text summarized in a more concise format to obtain better readability.

Our tool will first create send and reply Messages for `new C(int)`. The send Message will be immediately inserted into the main Interaction while the reply will be placed into a reply queue. Next `new A()` is processed. Now the tool needs to determine whether `new A()` is part of execution of `new C(int)` or if `new C(int)` has finished executing. `new A()` is called from the default constructor `new B()`, rather than the `new C(int)`.

Because the last message in the queue is a constructor, we compare the receive Lifeline of the last Message in the reply queue (`new C(int)`) and the send Lifeline of the current Message (`new A()`). If their Lifelines match then we know that `A()` execution is contained inside `C(int)` execution. `C(int)`'s reply Message is left in the queue, two Messages are created for `A()` and of those two, the reply Message is placed in the reply queue, waiting for the processing of `A()` to finish.

Next, `simpleMultiplication(int)` is processed. By confirming that the signature of the last Message in the reply queue (`new A()`) matches the method containing the current Message call (`simpleMultiplication()`) we determine that `simpleMultiplication()` call is made inside of the `new A(int)` execution. Nothing is removed from the reply queue and the reply Message for `simpleMultiplication(int)` is added to the queue.

The following Message to be processed is `ifAndFor()` (Figure 42). Its containing method signature is compared against the three Messages currently in the reply queue starting with the last one queued. None of the signatures match so it can be deduced that `simpleMultiplication()`, `new A()` and `new C(int)` have completed their execution. Their reply Messages are removed from the reply queue and placed in the Interaction before the Messages for `ifAndFor()` are created.

Figure 42 – Scenario diagram for the running example, part 2

Figure 42 contains an example of a nested combined fragment. Just before the `new D()` message is created, the check is made whether this method call is contained inside any condition in the control flow graph. Since `new D()` is inside a condition (`attr == 0`), a check is made whether the last message in the reply queue is inside a combined fragment corresponding to this condition. Since this is not true, a new combined fragment is created for this condition and the current Message, `new D()` is placed inside it. The

next Message, `callToA()` is inside the same combined fragment as `new D()` but also two more (loop and else). The following Message, `threeCalls(B)` is not inside of any conditions in its control flow graph, but because it is inside the execution of `callToA()` it is automatically placed inside of the same combined fragments as `callToA()`. The guards for the loop are set as `[1,1]` because in the scenario diagram we display a single traversal of the loop at a time. The visualization tool we use here, RSA, does not handle this correctly hence `[0,*]` is displayed, this is a small detail and can be changed manually.

Figure 43 – Scenario diagram for the running example, part 3

Figure 43 displays the next iteration of the loop started in Figure 42. This time, execution is passing the if, rather than the else section of the if-else condition in the `ifAndFor()` method. In a UML sequence diagram, rather than a scenario diagram, these two sections would be joined into an `alt` combined fragment.

Figure 44 – Scenario diagram for the running example, part 4

Figure 44 displays the third iteration of the loop, which makes the same method calls as the first one. Note that our default way to deal with loops is to check whether the subsequent Message is from a lower line number than the previous one if both are inside the same loop. This will flag that the loop has restarted. This is not as obvious when one iteration of the loop goes through an if condition and the next iteration goes through an else. For this purpose, we keep track of the smallest and largest line numbers for each alt combined fragment and check that, if we are executing an “else” method call, a previous method call was not inside an “if”. In Figure 44, “else” is executed following “if” (Figure

43) and it was the line number check that recognized that the loop needs to be interrupted for another iteration.

Figure 45 – Scenario diagram for the running example, part 5

The fourth iteration of the loop, shown in Figure 45, contains one method call more than previous iterations. This is because during this iteration, unlike before, condition `(b.attr > 0)` is true and the method call `simpleMultiplication()` inside `threeCalls(B)` method is executed.

The scenario diagrams presented in Figure 41 through Figure 45 are equivalent to the manually created sequence diagram from Figure 8, but there are some differences apart from the already discussed issue of the loop being presented separately for each iteration rather than as one whole.

The create Messages in Figure 8 are stand-alone and do not have behaviour execution specification bars representing their execution. Create Messages in the scenario diagrams are represented as synchronous messages. This is because our approach relies on the existence of reply Messages to determine when any particular method has finished executing. As a result we specify constructor Messages as synch rather than create Messages. One of the side effects of that is that all of the Lifelines in the scenario diagram start at the top of the diagram and not as their respective create messages reach them as is the case of Lifeline D in Figure 8.

6 CASE STUDY

To verify our hypothesis and analyze the results we are going to pose three research questions in section 6.1. Section 6.2 details the programs we used to validate our approach and section 6.3 gives details on how these programs will be used to obtain results and analyze performance of our approach. More details on how experimental execution time is measured are given in 6.4 and the results are discussed in 6.5.

6.1 Research questions

The objective of our study is to reduce the instrumentation overhead with respect to Leduc’s thesis while not losing visualization detail. Our research questions are as follows:

1. Is the execution overhead, measured as execution time, reduced when our approach is used?
2. Are the resulting scenario diagrams correct?
3. Are the resulting scenario diagrams equivalent to Leduc’s results?

6.2 The case study systems

We selected four different programs for our experiments. Some characteristics of the case study systems are summarized in Table 1. The table also indicates which case study system is used to answer which research question.

The first program is the “Example” program we used in Chapter 1. This example is simple and we can control how many method calls occur during program execution through input variable value. The input takes an integer which then determines the number of times the loop (line 10 in Figure 5) is run. One sequence diagram resulting from the analysis of this program (using one execution) has already been discussed in section 5.6.

The second program, “PCC Prover” is a course project for SCI 5110, a graduate level computer science course. It uses a Proof Carrying Code technique [17] to verify that a given code is safe to use. The program in this case study takes two text files, one with the source code to be verified and the other with the safety rules against which the source code will be tested. This program is convenient for our study because we can control the length of execution by changing the contents of the two input files.

The third program, “Calculator” is a simple calculator (partly generated with JavaCC) that uses the Visitor design pattern. We can confirm the implementation against a general design pattern. In other words, we have an oracle for deciding whether the generated, rever-engineered sequence diagram is correct or not. For our experiment, we asked the calculator to tell us the result of “1+1”.

The fourth program, “Library”, is the one used as a case study in [2]. This system relies heavily on user input so it will not be used to examine execution time because this would require us to modify the source code by inserting timer for the use cases we wanted to verify. While this is feasible, we have a sufficient number of other programs whose execution time we can analyze so we prefer to use this system to investigate whether the reverse-engineered sequence diagrams using our approach and Leduc’s are equivalent. The resulting sequence diagram will be analyzed with respect to discussion in [2].

Case study name	Number of classes	Number of methods	Number of Lines of code	Research questions
Example	4	13	56	1
PPC Prover	8	59	1280	1
Calculator	16	130	1175	1, 2
Library (Server)	44	459	3280	2, 3

Table 1 – Case studies characteristics

Table 1 gives a brief overview of the size of the systems we study. Example is the smallest with 4 classes and Library is the largest.

6.3 Test cases

6.3.1 Test case setup

To execute our running example we set the input integer to 200. This will cause the loop in the program to execute 200 times.

The PCC Prover takes two text files as input. The text files used here included a generic safety rule set that was packaged with the project. We created the “code” file by writing two lines (ADD r1 := r3 + r3, RET). This is considered an extremely small example of code. This will result in a very brief execution relative to the most common input for this program. The execution is kept small because a large number of object is created during execution. During preparation for the experiment we run the program with input files that were packaged with the project and the resulting Trace.txt file was too large for the operating system to open. This is an issue that will have to be addressed in future work.

For our experiment with the Calculator program, we asked for the result of “1+1”.

Because our tool does not support RMI, during the Library System study, only the server side is instrumented. No other modification is made to the system. Both server and client were launched and the following actions (scenario / test case) were taken:

- Log in as administrator
- Create senior librarian account
- Logout
- Log in as senior librarian
- Attempt to add copy before title is added (the attempt will fail)
- Add title
- Add copy

Though the entire interaction was captured in the original log, we will only discuss the last step (Add copy) when discussing the results. The Trace.txt contains only method call logs generated by the server.

6.3.2 Execution overhead

To analyze execution overhead, programs “Example”, “PCC Prover” and “Calculator” will be used. For each program we will prepare three versions: One with no instrumentation (base), one with our (light) instrumentation, and one with the (original) instrumentation used by Leduc [2].

The original instrumentation contained information about node ID and timestamp for every instrumentation call because it handles multithreading and multinode communication. Because the light instrumentation does not support these features, the timestamp and node ID calls were removed from the original instrumentation and the case study systems are not distributed or multi-threaded ones to make instrumentation calls between the two instrumentation techniques comparable.

6.3.3 Execution inputs and repeated executions

The use case “Add copy” from the Library system discussed in [2] has been reproduced using our tool. This use case has been chosen because it has been already examined in detail by Leduc, so instead of comparing the results produced by our approach to the incomplete design documentation, we will discuss discrepancies between the output produced by original instrumentation, shown in Figure 46 and the output created by our tool.

Figure 46 – Scenario diagram produced with the original tool [2]

Note that the Leduc's tool handles RMI and that this Add Copy use case demonstrates interaction between a client (Library Employee) and server (Library Server) by having the client initiate the execution in the use case. Since our tool does not support RMI, the section of the diagram representing the client will not exist in our result. The only difference between our diagram and Leduc's should then be the identification of node identifiers in the diagram and lack of RMI communication.

6.4 Measurement

Capturing execution times when evaluating execution overhead was accomplished by preparing a .bat file that calls each version of the program 100 times and is executed twice at two separate occasions. Each of the execution scenario discussed in the previous section is therefore executed 200 times. This is control for the possible impact of the Windows operating system. (When possible, all other applications running on the computer are turned off, and the network was disconnected.) Between calls to the program under study, there is a call to a timer to get start and end times of the execution. The time printouts are then imported into a spreadsheet and analyzed.

```

1      echo off
2      cd ..
3      cd PCC_original\bin
4      java -cp C:\Users\owner\workspace\Time\bin time.Time Start
        PCC_original
        java pcc.InvisiblePCC
        "C:\\Users\\owner\\workspace\\CaseStudyBatFiles\\Code4.txt"
        "C:\\Users\\owner\\Documents\\sci5110\\SCI5110\\PCCSystem_1\\
        Rules\\UserRules.txt"
5      java -cp C:\Users\owner\workspace\Time\bin time.Time End
        PCC_original,
6      cd ..\..

```

Figure 47 – Excerpt of the .bat file used to run experiment

Figure 47 displays an excerpt of the .bat file used in the experiment to execute the uninstrumented version of the PCC Prover program. The .bat file sets non verbose mode (line 1) and ensures that it is in the correct directory (lines 2 and 3).

```

1      package time;
2      public class Time {
3          public static void main(String[] args) {
4              System.out.println( args[0] + "," + args[1] + "," +
                System.currentTimeMillis());
5          }
6      }

```

Figure 48 – Class Time

Then it executes a call to a java program `time.Time`, Figure 48, that will print the current time (in milliseconds) as well as the passed string, in this case “Start PCC_original” (line 4). Class `Time` uses the JDK `System.currentTimeMillis()` method to collect the current machine time. The .bat file then executes the test case (line 4) by invoking `pcc.InvisiblePCC` with the two input text file names as parameters. Finally, the `time.Time` java program is called again (line 5), this time with parameters indicating “End” and an extra comma. When the output data is imported into a spreadsheet, the additional comma in the “End” time line compared to “Start” will cause the end time value to be imported into a different column therefore making it easy for the spreadsheet program as well as the person visually surveying the data to quickly and accurately differentiate beginning times from end times.

```
C:\Users\owner\workspace\CaseStudyBatFiles>echo off
Start,PCC_original,1286474599985
End,PCC_original,,1286474619392
Start,PCC_original,1286474619563
End,PCC_original,,1286474638923
```

Figure 49 – Excerpt of the experimental output

Figure 49 displays an excerpt of the output produced by the above .bat file. From this output, by subtracting start time from the end time, we determine the execution time of each execution.

In addition to execution time, the number of method calls made during execution will be used to analyze the data. Though data in Table 1 provide a good idea of the size and complexity of the programs we are studying, the number of actual method calls made during executions will be used during the analysis. These numbers are more accurate than general (static) information about the program because they are directly related to how many instrumentation calls are made. There is no experimental set-up to capture these numbers, they will be obtained separately (offline) from the execution trace files.

Since all executions will be done on the same computer, and we repeat executions many times to control for external perturbations, execution time (actually a mean execution time) is a good measure of the burden different instrumentation techniques place on the system.

6.5 Experimental results

6.5.1 Execution time

Table 2 lists the number of method calls and instrumentation calls for each program while execution times are displayed in Figure 50.

Program	Method calls executed	"Light" Instrumentation calls	"Original" Instrumentation calls
Calculator	113	131	305
PCC Prover	1138	1277	2334
Example	10000	10003	32000

Table 2 – Number of method calls for each program

We expect that the number of instrumentation calls has a strong impact on the execution time as these calls include writing to a file, amongst other computations. For the light instrumentation, the number of instrumentation is the sum of the number of method calls and the number of constructors executed. For the original instrumentation, the number of method calls is the sum of twice the number of method calls and twice the number of conditions and loops encountered.

Figure 50 – Execution time vs. instrumentation level by program

The experimental results are presented in Figure 50. The figure shows box plots, indicating minimal and maximal execution time values (opposing ends of the vertical line) as well as first and third quartile lines encompassing time range achieved by half of the total executions. While we tried to control other operating system activities, there is still a large variation in the execution times obtained. However, compared to the differences between minimum and maximum execution times for all 9 variations, most execution times lie within a narrow range. It is notable that the largest number of points for each variation is found at or near the minimum execution time. This is likely due to the fact that most of the time there were very few other processes running on the computer. The higher points probably occurred during times that the processor was handling another system event.

The variation between the fastest time and the median is noticeable for Calculator light, PCC Prover original and Example original. We attribute this to the instrumentation making system calls to write to file. Because these calls are external to the java execution, they may be more susceptible to external processes. For executions over 1000 method calls (PCC Prover and Calculator) the number of instrumentation calls is significant and it is more than double for original than for light versions. For Calculator, the light version is more negatively affected by instrumentation, than the original despite lower number of instrumentation calls. This could be because the amount of characters written into a file for light instrumentation is higher than for original. For example, the first trace entry for Calculator light was 249 characters, including empty spaces versus 175 for original. Light instrumentation is therefore not “lighter” than original for executions small enough not to be negatively affected by large number of instrumentation calls.

In the remainder of the analysis median value will be used to represent execution time for each of the nine variations.

Figure 51 – Median execution time vs. method calls executed

The experimental results (Figure 51) confirm the expectations that the light instrumentation approach presented here will, on a large enough program, causes the program to execute slower than it would without instrumentation but faster than with the original instrumentation approach. For programs with very few method calls both light and original instrumentations add about the same amount of delay, though as mentioned earlier, original instrumentation might have a slight advantage due to the fact that less information is stored in the traces. The presence of instrumentation requires a file to be created and written to, which are execution-heavy tasks (much heavier than any instrumentation-related behaviour added to the programs). Additionally each call to instrumentation will execute a write. Even though the original instrumentation writes to this file at least twice as often as our light instrumentation, the number of times the file is written to is small so the difference at this stage is not present. We suspect that the overhead over the non instrumented execution when the program is small mostly comes from creation of the trace file.

As the size of the program increases, which we simulate here by the increase in number of method calls, the execution time difference increases with it and the fact that light instrumentation makes less than half of the instrumentation calls of the original approach becomes more pronounced. The program execution time for the non-instrumented cases does not increase significantly which underscores the computational cost of writing to a file. Other log storage mechanisms should be investigated in the future.

6.5.2 Class size

The class size was acquired by listing all the files in the bin directory with the “dir” command from the command prompt. For each project (base, light, original) we summed up the class size for the regular program files and added instrumentation files separately. The results are summarized in Figure 52. The bottom of the bar shown in blue represents the size of the program files (files that existed in the project prior to addition of instrumentation) after they have been compiled with instrumentation. Top part of the bar, shown in red is the size of the compiled instrumentation files. Base version of each program only has the blue portion of the bar because no instrumentation was added.

Figure 52 - Impact of instrumentation on class size

Both instrumentation approaches (light and original) made a significant impact on the compiled class size compared to the uninstrumented size (base). The difference between original and light instrumentation is less pronounced but still significant, especially as the program size increases. For all three systems, the size of both regular program classes and summed instrumentation classes was larger for original than for light. In all cases each program class was larger for the original instrumentation. For instrumentation classes, `MethodAspect.aj`, `IdentificationAspect.aj` and `ObjectIdentification.java` tended to be a bit larger for light instrumentation because the source of these files is larger for light, but the light `Logger` was smaller and the addition of the two aspect files concerning control flow instrumentation caused the sum of instrumentation classes to always be larger for original than light. Overall, the light instrumentation strategy shows improvement over the original as far as compiled class size is concerned.

6.5.3 Scenario diagram visualization

6.5.3.1 Calculator

To answer the question whether the scenario diagram obtained by our tool is correct, we are going to consider the scenario diagram resulting from the Calculator execution. From the documentation and source code we know that the Calculator is an implementation of a Visitor pattern and that it traverses the math question (in our case “1+1”) in the form of an abstract syntax tree (AST).

The Visitor pattern allows the definition of new operations (visitors) without changing the classes or elements on which it operates (i.e., the data).

We have obtained the trace by running the Calculator. Because for our purpose we are not interested in how the tree is built, but only how it interacts with the visitor, we deleted all trace statements prior to the `SumVisitor` constructor call from the trace file. The tool was run and the results displayed in Figure 53 and Figure 54.

Figure 53 – Scenario diagram for Calculator – part 1

Figure 54 – Scenario diagram for Calculator – part 2

The AST is clearly visible in the scenario diagram. `ASTExpression (1+1)` has one child, `ASTOperator (+)` who has two children, two `ASTOperands (1)`. The visitor pattern is likewise clearly visible. After the `SumVisitor` object is created (first Message in Figure 53), the `accept` method is invoked on `ASTExpression` and the `Visitor` object is passed as parameter.

6.5.3.2 Library System

We proceeded to obtain the scenario diagram for the “Add Copy” use case by processing the trace file of the server part of the Library System. In order to obtain the diagram discussed by Leduc, we removed traces occurring before the `addCopy` button is selected by the user on the Library interface. The traces we removed had to do with the

initialization of the server and logging in the user as well an earlier unsuccessful attempt to add a copy and addition of a title. The trace was processed by the tool and the UML XMI file produced. The diagram, while covering the same use case still had more Messages. These Messages are the internal to methods called whose executions were visible in [2]. Like Leduc, we chose not to display this diagram in here because of its size (over 20 pages). Figure 55, Figure 56 and Figure 57 display the resulting diagram without the messages not visible in [2].

Figure 55 – Scenario diagram for Add Copy use case, part 1

In Figure 55 the first message, getTitle(String) corresponds to the third message getTitle("123-4567") in Figure 46. The remaining messages in Figure 55 line up sequentially with messages in Figure 46. The “Opt” combined fragment starts in Figure 55 and encloses all the following messages, except the last “reply”, in the remaining Figures.

Figure 56 - Scenario diagram for Add Copy use case, part 2

In Figure 56 message `new server.Copy(String, long)` corresponds to message `Create("123-4567",1)` in Figure 46. Execution of the constructor containing `setCopyOf(String)` and `setBarcode(long)` follows, as well as `addCopy(Copy)`, `getbarcode()` and `getLongestPendingReservation()` messages.

Figure 57 - Scenario diagram for Add Copy use case, part 3

Figure 57 displays the last three full messages and closure of the combined fragment.

Note that the message labels and Lifeline labels have a different format than in Leduc's diagram in Figure 46. These are stylistic differences due to our algorithm and our use of Rational Software Architect (RSA) tool to display the diagram. Furthermore, our algorithm does not handle RMI, so notions such as Node 0 and Node 1 are not applicable. In Figure 55 `EmployeeControl` does not receive the `addCopy` message because this message arrives from a different node and is therefore not in our trace.

Figure 55, Figure 56 and Figure 57 display an `opt` combined fragment representing the condition "`t1!=null`" which was explicitly stated for every affected message in Figure 46. This difference is due to the fact that Leduc used UML version 1.x (which does not have a notion of combined fragment) whereas we have combined fragments in UML 2.1 (the version we use).

Overall we find that the execution overhead, measured as execution time, is reduced when our approach is used. Except for very small executions (less than 1000 method calls executed), the light approach yielded better results both in terms of faster execution time and less variability from the minimum execution time. Based on analyzing the example of an execution using a visitor pattern and the Library case study previously discussed in

[2], we are satisfied that the scenario diagrams produced are correct and equivalent to Leduc's scenario diagrams.

7 CONCLUSION

In this thesis, we have presented a methodology for reverse engineering source code and generate scenario diagrams, which we defined as an intermediate step towards generating standard UML sequence diagrams, using a combination of static and dynamic analysis. The approach and tool we presented are built for Java programs, but they should be easily extended to other similar platforms such as python, ruby or smalltalk, and perhaps C++ (C++ has a much more complex syntax to account for). We used widely accepted AOP techniques for instrumentation, formally specified our intermediate models, used third party model transformation technology and represented our output in XMI format representing UML constructs. All this makes our work easily formally comparable and repeatable.

Our goal was to reduce the impact of the instrumentation from previous study by combining dynamic with static analysis. This was accomplished by both completely avoiding any modification (instrumentation) of the source code and reducing the amount of AOP instrumentation for tracing method executions. We evaluated the execution time of the original approach and our lightweight approach and indeed found a significant reduction in overhead was achieved using our method.

To evaluate the quality of the scenario diagrams created using our lightweight approach we analyzed a scenario diagram from a program implementing a Visitor pattern and reproduced a Library Case Study from [2]. It was found that no information was lost and that our approach is satisfactory.

8 FUTURE WORK

This section discusses issues that can be addressed when continuing the research presented in this document.

One of the premises suggested in this thesis is that Leduc's technique for reverse engineering concurrent and distributed systems could be easily reused with the approach presented in this thesis. This ought to be implemented. Node ID and thread ID would need to be collected through the Leduc's existing instrumentation approach and possibly additional instrumentation, in particular concurrency and RMI aspects would be added. Furthermore, the trace model would have to be extended to handle the additional instrumentation information (e.g., node ID, thread ID). There is however no reason to believe that this is unfeasible. We do not foresee any difficulty because of the simplicity of Leduc's mechanisms. Our algorithm relies on single threaded synchronous messages when location of the reply message is calculated. This would be more complicated when asynchronous communication from concurrent executions is introduced and we may have to partially resort to instrumenting the end of method executions as Leduc did. However, since this instrumentation would only be partial (confined to interfaces between executions) and since we do not instrument conditions and loops at all, we believe that the overall performance in terms of overhead reduction compared to Leduc's tool would not be significantly reduced.

Once concurrency and RMI are introduced into the tool, future work discussed in Leduc's thesis such as detection of signals and handling of exceptions and lost remote calls can be attempted.

While our approach is lighter in terms of execution overhead, the algorithm to compute scenario diagram from static and dynamic data is more complex than what Leduc uses to map trace to scenario diagram. It would be interesting to study the impact of increased algorithm complexity on transformation execution time.

Another aspect to address in the future is that the control flow parser we use is incomplete as described in Chapter 3. This parser should be completed to cover the entire Java grammar. We do not foresee any difficulty since we already support the majority of the control flow structure Java provides.

Since future implementation of this approach would not be created with the intent of comparing it in detail to Leduc, there is more freedom in how to implement aspects. For example, the `Logger` may be changed to keep all logs in memory and write to file at the end of the execution (i.e., only one disk access instead of many) to improve efficiency.

Our models were stored in text files. This was sufficient for the size of programs we analyzed, but would not be appropriate when analyzing large systems for which a database might be more appropriate. Also, a standard format such as CTF [12] may be employed.

The most interesting problem, however, would be to transform the scenario diagrams we produce into UML sequence diagrams. We would need to be able to condense multiple traversals of a single loop into one combined fragment and combine several scenario diagrams of the same use case into one diagram (one overarching Interaction). One way to do this is to zero in on one operation at the time. In a scenario diagram, one or more behaviour execution specifications are associated with an operation. All these behaviour execution specifications, and their associated sequence of messages, are representing execution of the same method. By analyzing these sequences of messages, while referring back to the control flow graph and through the use of a pattern matching algorithm, we should be able to construct a sequence diagram for the method execution and store it in the overarching interaction as an interaction use. Once all operations have been processed, we can reconstruct the overarching interaction.

The next challenge is handling overloading of methods as messages from the scenario diagram now summarized in a single message in the sequence diagram could be making references to different Interaction Uses. To handle this, we can place all relevant

Interaction Uses inside of an “alt” Combined Fragment indicating that the choice of Interaction binding is decided at run time.

Our resulting sequence diagram, though containing accurate representation of the execution, would look very awkward since Messages always invoke Interaction Use rather than the more readable Behaviour Execution Specification. The sequence diagram can then be modified again, depending on the size of the diagram and user’s preferences.

9 REFERENCES

- [1] IBM Corporation, Rational Software Architect, <http://www.ibm.com/developerworks/rational/products/rsa/>, 30 Oct. 2005, Web, 12 Jan. 2011.
- [2] Johanne Leduc, "Towards Reverse Engineering of UML Sequence Diagrams of Real-Time, Distributed Systems through Dynamic Analysis", M.A.Sc Thesis, Carleton University, Ottawa, 2004.
- [3] Joseph D. Gradecki and Nicholas Lesiecki, "Mastering AspectJ: aspect-oriented programming in Java", *Wiley*, 2003.
- [4] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis", *Software Engineering, IEEE Transactions on* , vol.35, no.5, Sept.-Oct. 2009
- [5] International Conference on Software Maintenance (ICSM), <http://conferences.computer.org/icsm/>. 28 Oct. 2010, Web, 12 Jan 2011.
- [6] European Conference on Software Maintenance and Reengineering (CSMR), <http://www.csmr.eu/>, n.d., Web, 12 Jan. 2011.
- [7] Model Driven Engineering Languages And Systems (MoDELS), <http://researchr.org/conferenceseries/models>, n.d., Web, 12 Jan. 2011.
- [8] IEEE International Conference on Program Comprehension (ICPC), <http://www.program-comprehension.org/>, 20 Sep. 2010, Web, 12 Jan. 2011.
- [9] B.A. Malloy and J.F. Power, "Exploiting UML Dynamic Object Modeling for the Visualization of C++ Programs", *Proc. ACM 2005 Symp. Software Visualization*, 2005.

- [10] Anthony Cleve and Jean-Luc Hainaut, "Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering", *15th Working Conference on Reverse Engineering*, 2008.
- [11] M. H. Alalfi, J. R. Cordy, T. R. Dean, "Automated Reverse Engineering of UML Sequence Diagrams for Dynamic Web Applications", *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*. IEEE Computer Society, Washington, DC, USA, 287-294, 2009.
- [12] Abdelwahab Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension", Ph.D Thesis, University of Ottawa, Ottawa, 2005.
- [13] MaintainJ, <http://www.maintainj.com/index.html>, n.d., Web, 12 Jan. 2011.
- [14] reverseJava, <http://www.reversejava.com/reversejavahome.htm>, 15 Apr. 2010, Web, 12 Jan. 2011.
- [15] JSonde, <http://www.jsonde.com/>, n.d., Web, 12 Jan. 2011.
- [16] Javacalltracer, <http://sourceforge.net/projects/javacalltracer/>, n.d., Web, 12 Jan. 2011.
- [17] JavaCC Parser Generator, <http://java.net/projects/javacc/>, 17 Dec. 2010, Web, 12 Jan. 2011.
- [18] OMG, "Unified Modeling Language (OMG UML), Superstructure, V2.1.2", 2007, <http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF>, n.d., Web, 12 Jan. 2011.
- [19] MDWorkbench, <http://www.mdworkbench.com/>, 19 Sep. 2009, Web, 12 Jan. 2011.
- [20] Proof Carrying Code, <http://www.eecs.berkeley.edu/~necula/pcc.html>, 22-Jul-2002, Web, 12 Jan. 2011.

Appendix A Aspect files for the running example

```
package example;
import java.util.*;
public aspect IdentifierAspect {
public int A.objectID = A.objectIDgenerator();
private static int A.currentObjectID = 1;
private static int A.objectIDgenerator() {
    LinkedList <String> log = new LinkedList <String> ();
    log.add("<lifeline className=\"example.A\" name=\"A_\" + A.currentObjectID
+ \"\"/>");
    Logger.getLoggingClient().instrument(log);
    return A.currentObjectID++;
}
declare parents : A implements ObjectID;
public String A.getObjectID() {
    String s = "A_" + objectID;
    return s;
}
public int B.objectID = B.objectIDgenerator();
private static int B.currentObjectID = 1;
private static int B.objectIDgenerator() {
    LinkedList <String> log = new LinkedList <String> ();
    log.add("<lifeline className=\"example.B\" name=\"B_\" + B.currentObjectID
+ \"\"/>");
    Logger.getLoggingClient().instrument(log);
    return B.currentObjectID++;
}
declare parents : B implements ObjectID;
public String B.getObjectID() {
    String s = "B_" + objectID;
    return s;
}
public int C.objectID = C.objectIDgenerator();
private static int C.currentObjectID = 1;
private static int C.objectIDgenerator() {
    LinkedList <String> log = new LinkedList <String> ();
    log.add("<lifeline className=\"example.C\" name=\"C_\" + C.currentObjectID
+ \"\"/>");
    Logger.getLoggingClient().instrument(log);
    return C.currentObjectID++;
}
declare parents : C implements ObjectID;
public String C.getObjectID() {
    String s = "C_" + objectID;
    return s;
}
}
public int D.objectID = D.objectIDgenerator();
private static int D.currentObjectID = 1;
private static int D.objectIDgenerator() {
    LinkedList <String> log = new LinkedList <String> ();
    log.add("<lifeline className=\"example.D\" name=\"D_\" + D.currentObjectID
+ \"\"/>");
    Logger.getLoggingClient().instrument(log);
    return D.currentObjectID++;
}
declare parents : D implements ObjectID;
public String D.getObjectID() {
    String s = "D_" + objectID;
    return s;
}
}
```

Figure 58 – IdentifierAspect.aj


```

package example;
import java.util.*;

public aspect MethodAspect {

    pointcut callMethod() : call (* *.*(..)) && 'call (static * *.*(..)) && 'call
(* example.IdentifierAspect.*(..))&& 'call (* example.Logger.*(..)) && 'call (*
example.ObjectID.*(..)) ;
    pointcut callStaticMethod() : call (static * *.*(..)) && call (*
example.*.*(..)) && 'call (* example.MethodAspect.*(..)) && 'call (*
example.IdentifierAspect.*(..)) && 'call (* *.*.objectIDgenerator(..)) && 'call (*
example.Logger.*(..));
    pointcut callConstructor() : call (*.*.new(..)) && 'call (example.Logger.new(..));

    before(): callMethod () {
        String thisID = new String ();
        LinkedList log = new LinkedList ();
        if (thisJoinPoint.getThis() != null) {
            thisID = String.valueOf(((ObjectID)
thisJoinPoint.getThis()).getObjectID());
        }
        else {
            thisID =
getStaticClassName(thisJoinPointStaticPart.getSourceLocation().toString());
        }
        String targetID = new String ();
        targetID = String.valueOf(((ObjectID)
thisJoinPoint.getTarget()).getObjectID());
        log.add("<messageLog bindToClass=\""
+
MethodAspect.getBindToClassName(thisJoinPoint.getTarget().toString()) + "\"
messageSort=\"synchCall\" signature=\""
+ MethodAspect.getMethodSignature(thisJoinPoint.toString()) +
"\">>");
        log.add("    <sendEvent covered=\"" + thisID + "\"/>");
        log.add("    <receiveEvent covered=\"" + targetID + "\"/>");
        log.add("    <sentFrom lineNumber=\""
+
MethodAspect.getLineNumber(thisJoinPointStaticPart.getSourceLocation().toString()) + "\"
name=\""
+
MethodAspect.getFileName(thisJoinPointStaticPart.getSourceLocation().toString()) +
"\">>");
        log.add("</messageLog>");
        Logger.getLoggingClient().instrument(log);
    }

    before(): callStaticMethod () {
        String thisID = new String ();
        LinkedList log = new LinkedList ();
        if (thisJoinPoint.getThis() != null) {
            thisID = String.valueOf(((ObjectID)
thisJoinPoint.getThis()).getObjectID());
        }
        else {
            thisID =
getStaticClassName(thisJoinPointStaticPart.getSourceLocation().toString());
        }
        log.add("<messageLog bindToClass=\""
+
MethodAspect.getStaticBindToClassName(thisJoinPoint.toString())
+ "\" messageSort=\"synchCall\" signature=\""
+
MethodAspect.getMethodSignature(thisJoinPoint.toString()) + "\"/>");
        log.add("    <sendEvent covered=\"" + thisID + "\"/>");
        log.add("    <receiveEvent covered=\""
+
getStaticLifelineName(thisJoinPoint.toString()) + "\"/>");
        log.add("    <sentFrom lineNumber=\""
+
MethodAspect.getLineNumber(thisJoinPointStaticPart.getSourceLocation().toString())

```

```

        + "\" name=\"" +
MethodAspect.getFileName(thisJoinPointStaticPart.getSourceLocation().toString()) +
"\"/>");
        log.add("</messageLog>");
        Logger.getLogger().instrument(log);
    }

    before(): callConstructor () {
        String thisID = new String ();
        LinkedList log = new LinkedList ();
        if (thisJoinPoint.getThis() != null) {
            thisID = String.valueOf(((ObjectID)
thisJoinPoint.getThis()).getObjectID());
        }
        else {
            thisID =
getStaticClassName(thisJoinPointStaticPart.getSourceLocation().toString());
        }
        log.add("<messageLog bindToClass=\"" +
MethodAspect.getNewBindToClassName(thisJoinPoint.toString())
        + "\" messageSort=\"createMessage\" signature=\"new \" +
MethodAspect.getMethodSignature(thisJoinPoint.toString()) + "\"/>");
        log.add("        <sendEvent covered=\"" + thisID + "\"/>");
        log.add("        <receiveEvent covered=\"nothing\"/>");
        log.add("        <sentFrom lineNumber=\"" +
        +
MethodAspect.getLineNumber(thisJoinPointStaticPart.getSourceLocation().toString())
        + "\" name=\"" +
MethodAspect.getFileName(thisJoinPointStaticPart.getSourceLocation().toString()) +
"\"/>");
        log.add("</messageLog>");
        Logger.getLogger().instrument(log);
    }

    private static String getLineNumber(String s){
        return s.substring(s.indexOf(":") + 1);
    }
    private static String getFileName(String s){
        return s.substring(0, s.indexOf(":"));
    }
    private static String getStaticClassName(String s){
        s = s.substring(0, s.indexOf("."));
        return s + "_static";
    }
    private static String getBindToClassName(String s){
        if (s.contains("@")) {
            s = s.substring(0, s.indexOf("@"));
        }
        return s;
    }
    private static String getStaticBindToClassName(String s){
        return s.substring(s.indexOf(" ") + 1, s.lastIndexOf("."));
    }
    private static String getNewBindToClassName(String s){
        return s.substring(s.indexOf("(") + 1, s.lastIndexOf("("));
    }
    private static String getMethodSignature(String s){
        return s.substring(s.indexOf("(")+1, s.lastIndexOf(")"));
    }
    private static String getStaticLifelineName(String s){
        if (s.contains("call(")){
            s = s.substring(5, s.length());
        }
        if (s.indexOf("(") >= 0){
            s = s.substring(0, s.indexOf("(") );
        }
        if (s.indexOf("@") >= 0){

```

```

        s = s.substring(0, s.indexOf("@") );
    }
    if (s.indexOf(".") >= 0){
        s = s.substring(s.indexOf(".") + 1, s.length());
    }
    return getStaticClassName(s);
}
}

```

Figure 59 – MethodAspect.aj

```

package example;

public interface ObjectID {
    public String getObjectID();
}

```

Figure 60 – ObjectID.java

Appendix B Control flow graphs for the running example

Figure 61 – Class A control flow diagram for the running example

Figure 62 - Class C control flow diagram for the running example

Figure 63 - Class D control flow diagram for the running example

Appendix C Trace and control flow models

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
xmlns:this="http://Trace/Trace.xsd" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://Trace/Trace.xsd">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      "Trace Metamodel Schema 07-sep-2010"
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType name="Log">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="messageLog"
type="this:MessageLog"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="MessageLog">
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="0" name="sendEvent"
type="this:MessageLogOccurrenceSpecification"/>
      <xsd:element maxOccurs="1" minOccurs="0" name="receiveEvent"
type="this:MessageLogOccurrenceSpecification"/>
      <xsd:element maxOccurs="1" minOccurs="0" name="sentFrom"
type="this:SourceLocation"/>
    </xsd:sequence>
    <xsd:attribute name="signature" type="xsd:string"/>
    <xsd:attribute name="messageSort" type="this:MessageSort"/>
    <xsd:attribute name="bindToClass" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="MessageLogOccurrenceSpecification">
    <xsd:attribute name="covered" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="SourceLocation">
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="lineNumber" type="xsd:int"/>
  </xsd:complexType>
  <xsd:simpleType name="MessageSort">
    <xsd:restriction base="xsd:NCName">
      <xsd:enumeration value="synchCall"/>
      <xsd:enumeration value="createMessage"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Figure 64 – Trace model specification

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
xmlns:this="http://ControlFlow/ControlFlow.xsd" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://ControlFlow/ControlFlow.xsd" xmi:version="2.1">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Control Flow metamodel schema
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType name="Class">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element ecore:opposite="methodClass" maxOccurs="unbounded" minOccurs="0"
name="method" type="this:Method"/>
    </xsd:sequence>
  </xsd:complexType>
```

```

<xsd:complexType name="Method">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="signature" type="xsd:string"/>
    <xsd:element ecore:opposite="method" maxOccurs="unbounded" minOccurs="0"
name="contains" type="this:CodeSection"/>
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="CodeSection">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <'-- <xsd:element ecore:opposite="contains" maxOccurs="1" minOccurs="0"
name="containingSection" type="this:ConditionalSection"/> -->
  </xsd:choice>
</xsd:complexType>
  <xsd:complexType name="ConditionalSection">
    <xsd:complexContent>
      <xsd:extension base="this:CodeSection">
        <xsd:sequence>
          <xsd:element name="conditionDescription" type="xsd:string"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="MethodCall">
    <xsd:complexContent>
      <xsd:extension base="this:CodeSection">
        <xsd:sequence>
          <xsd:element name="lineNumber" type="xsd:int"/>
          <xsd:element name="signature" type="xsd:string"/>
          <xsd:element name="isInMethod" type="xsd:string"/>
          <xsd:element name="isInClass" type="xsd:string"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Opt">
    <xsd:complexContent>
      <xsd:extension base="this:ConditionalSection">
        <xsd:sequence>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Alt">
    <xsd:complexContent>
      <xsd:extension base="this:ConditionalSection">
        <xsd:sequence>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Loop">
    <xsd:complexContent>
      <xsd:extension base="this:ConditionalSection">
        <xsd:sequence>
          <xsd:element name="loopType" type="this:LoopType"/>
          <xsd:element name="min" type="xsd:string"/>
          <xsd:element name="max" type="xsd:string"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="LoopType">
    <xsd:restriction base="xsd:NCName">
      <xsd:enumeration value="for"/>
      <xsd:enumeration value="do"/>
      <xsd:enumeration value="while"/>
    </xsd:restriction>
  </xsd:simpleType>

```

```
</xsd:restriction>  
</xsd:simpleType>  
</xsd:schema>
```

Figure 65 – Control flow model specification

Appendix D Trace, control flow graph for the running example

```
<messageLog bindToClass="example.C" messageSort="createMessage" signature="new
example.C(int)">
  <sendEvent covered="C_static"/>
  <receiveEvent covered="nothing"/>
  <sentFrom lineNumber="6" name="C.java"/>
</messageLog>
<lifeline className="example.B" name="B_1"/>
<lifeline className="example.C" name="C_1"/>
<messageLog bindToClass="example.A" messageSort="createMessage" signature="new
example.A()">
  <sendEvent covered="C_1"/>
  <receiveEvent covered="nothing"/>
  <sentFrom lineNumber="6" name="B.java"/>
</messageLog>
<lifeline className="example.A" name="A_1"/>
<messageLog bindToClass="example.A" messageSort="synchCall" signature="void
example.A.simpleMultiplication(int)">
  <sendEvent covered="A_1"/>
  <receiveEvent covered="A_1"/>
  <sentFrom lineNumber="16" name="A.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.C.ifAndFor()">
  <sendEvent covered="C_static"/>
  <receiveEvent covered="C_1"/>
  <sentFrom lineNumber="7" name="C.java"/>
</messageLog>
<messageLog bindToClass="example.D" messageSort="createMessage" signature="new
example.D()">
  <sendEvent covered="C_1"/>
  <receiveEvent covered="nothing"/>
  <sentFrom lineNumber="10" name="B.java"/>
</messageLog>
<lifeline className="example.D" name="D_1"/>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="int
example.B.callToA()">
  <sendEvent covered="C_1"/>
  <receiveEvent covered="C_1"/>
  <sentFrom lineNumber="16" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.A" messageSort="synchCall" signature="int
example.A.threeCalls(B)">
  <sendEvent covered="C_1"/>
  <receiveEvent covered="A_1"/>
  <sentFrom lineNumber="22" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.A" messageSort="synchCall" signature="void
example.A.emptyCall(String)">
  <sendEvent covered="A_1"/>
  <receiveEvent covered="A_1"/>
  <sentFrom lineNumber="8" name="A.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.B.quickCall()">
  <sendEvent covered="A_1"/>
  <receiveEvent covered="C_1"/>
  <sentFrom lineNumber="12" name="A.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.B.getD()">
  <sendEvent covered="C_1"/>
```

```

        <receiveEvent covered="C_1"/>
        <sentFrom lineNumber="13" name="B.java"/>
    </messageLog>
    <messageLog bindToClass="example.D" messageSort="synchCall" signature="int
example.D.get() ">
        <sendEvent covered="C_1"/>
        <receiveEvent covered="D_1"/>
        <sentFrom lineNumber="21" name="B.java"/>
    </messageLog>
    <messageLog bindToClass="example.C" messageSort="synchCall" signature="int
example.B.callToA() ">
        <sendEvent covered="C_1"/>
        <receiveEvent covered="C_1"/>
        <sentFrom lineNumber="16" name="B.java"/>
    </messageLog>
    <messageLog bindToClass="example.A" messageSort="synchCall" signature="int
example.A.threeCalls(B) ">
        <sendEvent covered="C_1"/>
        <receiveEvent covered="A_1"/>
        <sentFrom lineNumber="22" name="B.java"/>
    </messageLog>
    <messageLog bindToClass="example.A" messageSort="synchCall" signature="void
example.A.emptyCall(String) ">
        <sendEvent covered="A_1"/>
        <receiveEvent covered="A_1"/>
        <sentFrom lineNumber="8" name="A.java"/>
    </messageLog>
    <messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.B.quickCall() ">
        <sendEvent covered="A_1"/>
        <receiveEvent covered="C_1"/>
        <sentFrom lineNumber="12" name="A.java"/>
    </messageLog>

```

Figure 66 – Trace.txt contents for the running example

```

<?xml version="1.0" encoding="UTF-8"?>
<trace:Log xmlns:version="2.1" xmlns:xm1="http://schema.omg.org/spec/XMI/2.1"
xmlns:trace="http://www.mdworkbench.com/trace"
name="C:\Users\owner\workspace\Example3\Trace">
    <messageLog bindToClass="example.C" messageSort="createMessage" signature="new
example.C(int) ">
        <sendEvent covered="C_static"/>
        <receiveEvent covered="C_1"/>
        <sentFrom lineNumber="6" name="C.java"/>
    </messageLog>
    <messageLog bindToClass="example.A" messageSort="createMessage" signature="new
example.A() ">
        <sendEvent covered="C_1"/>
        <receiveEvent covered="A_1"/>
        <sentFrom lineNumber="6" name="B.java"/>
    </messageLog>
    <messageLog bindToClass="example.A" messageSort="synchCall" signature="void
example.A.simpleMultiplication(int) ">
        <sendEvent covered="A_1"/>
        <receiveEvent covered="A_1"/>
        <sentFrom lineNumber="16" name="A.java"/>
    </messageLog>
    <messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.C.ifAndFor() ">
        <sendEvent covered="C_static"/>
        <receiveEvent covered="C_1"/>
        <sentFrom lineNumber="7" name="C.java"/>
    </messageLog>

```

```

<messageLog bindToClass="example.D" messageSort="createMessage" signature="new
example.D()" ">
    <sendEvent covered="C_1"/>
    <receiveEvent covered="D_1"/>
    <sentFrom lineNumber="10" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="int
example.B.callToA()" ">
    <sendEvent covered="C_1"/>
    <receiveEvent covered="C_1"/>
    <sentFrom lineNumber="16" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.A" messageSort="synchCall" signature="int
example.A.threeCalls(B)" ">
    <sendEvent covered="C_1"/>
    <receiveEvent covered="A_1"/>
    <sentFrom lineNumber="22" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.A" messageSort="synchCall" signature="void
example.A.emptyCall(String)" ">
    <sendEvent covered="A_1"/>
    <receiveEvent covered="A_1"/>
    <sentFrom lineNumber="8" name="A.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.B.quickCall()" ">
    <sendEvent covered="A_1"/>
    <receiveEvent covered="C_1"/>
    <sentFrom lineNumber="12" name="A.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.B.getD(D)" ">
    <sendEvent covered="C_1"/>
    <receiveEvent covered="C_1"/>
    <sentFrom lineNumber="13" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.D" messageSort="synchCall" signature="int
example.D.get()" ">
    <sendEvent covered="C_1"/>
    <receiveEvent covered="D_1"/>
    <sentFrom lineNumber="21" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="int
example.B.callToA()" ">
    <sendEvent covered="C_1"/>
    <receiveEvent covered="C_1"/>
    <sentFrom lineNumber="16" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.A" messageSort="synchCall" signature="int
example.A.threeCalls(B)" ">
    <sendEvent covered="C_1"/>
    <receiveEvent covered="A_1"/>
    <sentFrom lineNumber="22" name="B.java"/>
</messageLog>
<messageLog bindToClass="example.A" messageSort="synchCall" signature="void
example.A.emptyCall(String)" ">
    <sendEvent covered="A_1"/>
    <receiveEvent covered="A_1"/>
    <sentFrom lineNumber="8" name="A.java"/>
</messageLog>
<messageLog bindToClass="example.C" messageSort="synchCall" signature="void
example.B.quickCall()" ">
    <sendEvent covered="A_1"/>
    <receiveEvent covered="C_1"/>
    <sentFrom lineNumber="12" name="A.java"/>
</messageLog>
</trace:Log>

```

Figure 67 – Trace.xmi contents for the running example

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:controlflow="http://www.mdworkbench.com/controlflow">
<controlflow:Class name="A">
  <method signature="example.A.emptyCall(String)">
    </method>
  <method signature="example.A.simpleMultiplication(int)">
    </method>
  <method signature="example.A.threeCalls(B)">
    <contains xsi:type="controlflow:MethodCall" lineNumber="8"
signature="emptyCall('arg') " isInMethod="example.A.threeCalls(B)" isInClass="A.java"/>
    <contains xsi:type="controlflow:Opt" conditionDescription="b.attr_GT_0">
      <contains xsi:type="controlflow:MethodCall" lineNumber="10"
signature="simpleMultiplication(b.attr)" isInMethod="example.A.threeCalls(B)"
isInClass="A.java"/>
    </contains>
    <contains xsi:type="controlflow:MethodCall" lineNumber="12" signature="quickCall()"
isInMethod="example.A.threeCalls(B)" isInClass="A.java"/>
    </contains>
  </method>
  <method signature="example.A.A()">
    <contains xsi:type="controlflow:MethodCall" lineNumber="16"
signature="simpleMultiplication(10)" isInMethod="example.A.A()" isInClass="A.java"/>
    </contains>
  </method>
</controlflow:Class>
<controlflow:Class name="B">
  <method signature="new B()">
    <contains xsi:type="controlflow:MethodCall" lineNumber="6" signature="new A()"
isInMethod="new B()" isInClass="B.java"/>
    </contains>
  </method>
  <method signature="example.B.B(int)">
    </method>
  <method signature="example.B.ifAndFor()">
    <contains xsi:type="controlflow:Opt" conditionDescription="attr==0">
      <contains xsi:type="controlflow:MethodCall" lineNumber="10" signature="new D()"
isInMethod="example.B.ifAndFor()" isInClass="B.java"/>
      <contains xsi:type="controlflow:Loop" conditionDescription="for (int i = 0;
i_LT_totalLoops; i)">
        <contains xsi:type="controlflow:Alt" conditionDescription="i==1">
          <contains xsi:type="controlflow:MethodCall" lineNumber="13" signature="getD(d)"
isInMethod="example.B.ifAndFor()" isInClass="B.java"/>
          </contains>
          <contains xsi:type="controlflow:Alt" conditionDescription="else i==1">
            <contains xsi:type="controlflow:MethodCall" lineNumber="16"
signature="callToA()" isInMethod="example.B.ifAndFor()" isInClass="B.java"/>
            </contains>
          </contains>
        </contains>
      </contains>
    </contains>
  </method>
  <method signature="example.B.getD(D)">
    <contains xsi:type="controlflow:MethodCall" lineNumber="21" signature="get()"
isInMethod="example.B.getD(D)" isInClass="B.java"/>
    </contains>
  </method>
  <method signature="example.B.callToA()">
    <contains xsi:type="controlflow:MethodCall" lineNumber="22"
signature="threeCalls(this)" isInMethod="example.B.callToA()" isInClass="B.java"/>
    </contains>
  </method>
  <method signature="example.B.quickCall()">
    </method>
</controlflow:Class>
<controlflow:Class name="C">
  <method signature="example.C.C(int)">
    </method>
  <method signature="example.C.main(String[])">
    <contains xsi:type="controlflow:MethodCall" lineNumber="6" signature="new
C(Integer.parseInt(arg[0]))" isInMethod="example.C.main(String[])" isInClass="C.java"/>
  </contains>
  </method>
</controlflow:Class>
</xmi:XMI>
```

```
<contains xsi:type="controlflow:MethodCall" lineNumber="6"
signature="parseInt (arg[0])" isInMethod="example.C.main(String[])" isInClass="C.java"/>
  <contains xsi:type="controlflow:MethodCall" lineNumber="7" signature="ifAndFor () "
isInMethod="example.C.main(String[])" isInClass="C.java"/>
</method>
</controlflow:Class>
<controlflow:Class name="D">
  <method signature="example.D.set (int) ">
</method>
  <method signature="example.D.get () ">
</method>
</controlflow:Class>
</xmi:XMI>
```

Figure 68 – ControlFlow.xmi contents for running example

Appendix E Model transformation code

```

package reverseengineering;

public ruleset reverseengineering(in sourceT : trace, in sourceCF : controlflow, out
target : uml21) {

    public rule main() {
        create model and collaboration
        model = target.create("Model");
        model.name = "Reverse Engineering Model";
        collaboration = target.create("Collaboration");
        collaboration.name = "Reverse Engineering Collaboration";
        model.packageElement.add(collaboration);

        take a set of all the ref of call <ows> in <to> ref of it is

        for each trace log create a interaction
        foreach (log : trace.Log in sourceT.getInstances("Log")) {
            interaction = target.create("Interaction");

            @createInteraction(log, interaction, model, collaboration);
            collaboration.ownedBehavior.add(interaction);

        } foreach log trace log in sourceT.getInstances("Log")
        System.out.println("Done!!!");
    }

    private rule createInteraction(log : trace.Log, interaction : uml21.Interaction,
        model : uml21.Model, collaboration : uml21.Collaboration){
        fragmentqueue = target.create("Interaction");
        messagequeue = target.create("Interaction");

        interaction.name = log.name;

        process message logs
        foreach (ml : trace.MessageLog in log.messageLog) {

            in the interaction each message log will map to a execution specification and
            related objects to form the execution specification of the message log to the execution
            specification

            for each message log determine if rule is already executed specification already
            exists

            get method call that maps to message log
            mc = sourceCF.getInstances("MethodCall").first();
            bFoundMC = false;

            foreach (iteratemc : controlflow.MethodCall in
                sourceCF.getInstances("MethodCall")) {

                if ((SignatureComparator.match(ml.signature.toString(),
                    iteratemc.signature.toString())) &&
                    (ml.sentFrom.name == iteratemc.isInClass) &&
                    (ml.sentFrom.lineNumber == iteratemc.lineNumber)){
                    mc = iteratemc;
                    bFoundMC = true;
                    break;
                }
            } foreach iteratemc controlflow.MethodCall in sourceCF
            check that method call was found

            if (bFoundMC == false)
            {
                System.out.println( "ERROR!!! Method Call was not found!");
                System.out.println( ml.sentFrom.lineNumber + " "

```

```

        + ml.sentFrom.name + " "
        + ml.signature.toString());
    }
    / create return messages

    @moveQueuedMessages(ml, mc, messagequeue, fragmentqueue);
    @openingMessage(ml, mc, messagequeue, fragmentqueue);
} // foreach "ml : trace.MessageLog in log.MessageLog"

@terminateQueuedMessages(messagequeue, fragmentqueue);
target.remove(messagequeue);
target.remove(fragmentqueue);

}

private rule createInteraction::openingMessage(ml : trace.MessageLog,
    mc : controlflow.MethodCall, messagequeue : uml21.Interaction,
    fragmentqueue : uml21.Interaction) {

    class = null;
    receiveClass = null;
    operation = null;
    sendProperty = null;
    sendLifeline = null;
    receiveProperty = null;
    receiveLifeline = null;
    connector = null;
    connectorend1 = null;
    connectorend2 = null;
    sendMessage = null;
    replyMessage = null;
    sendEvent = null;
    receiveEvent = null;
    createEvent = null;
    sendMOSSs = null;
    sendMOSr = null;
    replyMOSSs = null;
    replyMOSr = null;
    BES = null;
    enclosingInteraction = interaction;
    combFragment = null;
    innermostCombFragment = null;

    / find or create receive class
    bFound = false;

    foreach (itr : uml21.Class in target.getInstances("Class")) {
        if (itr.name == ml.bindToClass){
            receiveClass = itr;
            bFound = true;
            break;
        }
    }

    if (bFound == false){
        receiveClass = target.create("Class");
        receiveClass.name = ml.bindToClass;
        model.packagedElement.add(receiveClass);
    }

    / find or create operation
    bFound = false;
    foreach (itr : uml21.Operation in target.getInstances("Operation")) {
        if ((itr.name == ml.signature)
            && (itr.ownedComment.first().body == mc.isInClass + " {"
                + mc.lineNumber.toString() + "}")){
            operation = itr;

```

```

        bFound = true;
        break;
    }
}

if (bFound == false){
    operation = target.create("Operation");
    operation.name = ml.signature;

    lineNumberComment = target.create("Comment");
    lineNumberComment.body = mc.isInClass + " {"
+ mc.lineNumber.toString()
+ "}";
    operation.ownedComment.add(lineNumberComment);
    receiveClass.ownedOperation.add(operation);
}

find or create send property      and lifeline
bFound = false;
foreach (itr : uml21.Lifeline in target.getInstances("Lifeline")) {
    if (itr.represents.name == ml.sendEvent.covered){
        sendProperty = itr.represents;
        sendLifeline = itr;
        bFound = true;
        break;
    }
}

if (bFound == false){
    actor = target.create("Actor");
    actor.name = mc.isInClass;
    model.packagedElement.add(actor);

    sendProperty = target.create("Property");
    sendProperty.name = ml.sendEvent.covered;
    sendProperty.type = actor;

    collaboration.ownedAttribute.add(sendProperty);

    sendLifeline = target.create("Lifeline");
    sendLifeline.name = ml.sendEvent.covered;
    sendLifeline.represents = sendProperty;
    interaction.lifeline.add(sendLifeline);
}

find or create receive property    and lifeline
if (ml.receiveEvent.covered.compareTo(ml.sendEvent.covered) != 0){
    bFound = false;
    foreach (itr : uml21.Lifeline in target.getInstances("Lifeline"))
    {
        if (itr.represents.name == ml.receiveEvent.covered){
            receiveProperty = itr.represents;
            receiveLifeline = itr;
            bFound = true;
            break;
        }
    }
    if (bFound == false){
        receiveProperty = target.create("Property");
        receiveProperty.name = ml.receiveEvent.covered;
        receiveProperty.type = receiveClass;
        collaboration.ownedAttribute.add(receiveProperty);
        receiveLifeline = target.create("Lifeline");
        receiveLifeline.name = ml.receiveEvent.covered;
        receiveLifeline.represents = receiveProperty;
        interaction.lifeline.add(receiveLifeline);
    }
}

```



```

    } if (nl.receiveEvent.covered.compareTo(nl.sendEvent.covered) < 0)
    else {
        receiveLifeline = sendLifeline;
    }
    find or create connector and connector ends
    bFound = false;
    foreach (itr : uml21.Connector in target.getInstances("Connector")) {
        end1 = itr.end.first();
        end2 = itr.end.last();
        if (end1.role == sendProperty || end2.role == sendProperty){
            if((end1.role == sendProperty && end2.role ==
                receiveProperty)
                || (end2.role == sendProperty && end1.role ==
                receiveProperty)){
                connector = itr;
                bFound = true;
                break;
            }
        }
    }
    if (bFound == false){
        connector = target.create("Connector");
        end1 = target.create("ConnectorEnd");
        end1.role = sendProperty;
        end2 = target.create("ConnectorEnd");
        end2.role = receiveProperty;
        connector.end.add(end1);
        connector.end.add(end2);
        interaction.ownedConnector.add(connector);
    }
    create messages etc.

    closing interaction figure out if there is a nested call
    / for rest
    if (messagequeue.fragment.size() > 0) {
        find s that refer to this s
        bFound = false;
        foreach (itr : uml21.BehaviorExecutionSpecification in
            target.getInstances("BehaviorExecutionSpecification")) {

            if (itr.finish == messagequeue.fragment.last()){

                bFound = true;
                if (itr.enclosingOperand != null){
                    enclosingInteraction =
                        itr.enclosingOperand.owner;
                }
                else {
                    enclosingInteraction = interaction;
                }
                break;
            }
        }
        if (bFound == false){
            System.out.println( "ERROR!!! BES not found!");
        }
    }

    is method call inside a conditional section
    if (mc.containingSection != null) {
        breakLoop = 0;

        lastMessageLineNumber = 0;
        lastMessageClass = "";
        lastMessageIsInMethod = "";
    }

```

```

        if (interaction.message.last().ownedComment.size() > 1) {

            lastMessageLineNumber =
            Integer.parseInt(interaction.message.last().
            ownedComment[0].toString().substring
            (interaction.message.last().ownedComment[0].toString().indexOf("{")
            + 1,
            interaction.message.last().ownedComment[0].toString().indexOf("}")));
            lastMessageClass = interaction.message.last().ownedComment[0].
            toString().substring(0, interaction.message.last().
            ownedComment[0].toString().indexOf(" "));
            lastMessageIsInMethod = interaction.message.last().ownedComment[1].
            toString();
        }

        if (mc.isInClass.equals(lastMessageClass) && mc.isInMethod.equals(
            lastMessageIsInMethod)
            && ((mc.lineNumber < lastMessageLineNumber) ||
            (mc.lineNumber ==
            lastMessageLineNumber)
            &&
            ml.signature.equals(interaction.message.last().signature.toString())) {

            breakLoop = 1;

        }
        containingSection = mc.containingSection;

        while (containingSection != null) {

            find if the copied fragment already exists
            bFound = false;

            to see if the copied fragment already exists, first whether the previous section
            is already a fragment or not

            lookingforfragment =
            interaction.message.last().sendEvent.enclosingOperand.owner;

            if (lookingforfragment != null){
                if (lookingforfragment.operand.first() != null){
                    if
                    (lookingforfragment.operand.first().fragment.last() != null) {
                        if
                        (lookingforfragment.operand.first().fragment.last().getClass().toString().equals(
                            "class
                            com.sodius.mdw.metamodel.uml21.internal.impl.BehaviorExecutionSpecificationImpl")){
                            lookingforfragment = null;
                        }
                    }
                }
            }

            while ((lookingforfragment != null)
                && (!lookingforfragment.getClass().toString().equals(
                "class com.sodius.mdw.metamodel.uml21.internal.impl.Interaction"))){

                if (lookingforfragment.interactionOperator ==
                com.sodius.mdw.metamodel.uml21.InteractionOperatorKind.ALT_LITERAL

                && ((lookingforfragment.name.contains(containingSection.conditionDescription)
                && lookingforfragment.name.contains("else"))
                || (containingSection.conditionDescription.contains(lookingforfragment.name)
                && containingSection.conditionDescription.contains("else")))
                && !containingSection.conditionDescription.equals(lookingforfragment.name)
                && mc.isInClass.equals(lastMessageClass)

```

```

    && mc.isInMethod.equals(lastMessageIsInMethod)) {

        breakLoop = 1;
    }
    else if (containingSection.getClass().toString() ==
"com.sodius.mdw.metamodel.controlflow.impl.LoopImpl") {
        breakLoop = 0;
    }
    else if ((lookingforfragment.name ==
containingSection.conditionDescription)) {
        if (breakLoop == 0) {
            bFound = true;
            combFragment = lookingforfragment;
            if (containingSection ==
mc.containingSection) {
                innermostCombFragment =
                lookingforfragment;
            }
        }

        break;
    }

    lookingforfragment =
    lookingforfragment.enclosingOperand.owner;
    } while (lookingforfragment != null);

    if (bFound == false) {
        combFragment = target.create("CombinedFragment");
        if (containingSection == mc.containingSection) {
            innermostCombFragment = combFragment;
        }
        combFragment.name =
        containingSection.conditionDescription;
        if (containingSection.getClass().toString() ==
"com.sodius.mdw.metamodel.controlflow.impl.OptImpl") {
            combFragment.interactionOperator =
com.sodius.mdw.metamodel.uml21.InteractionOperatorKind.OPT_LITERAL;
        }
        else if (containingSection.getClass().toString() ==
"class com.sodius.mdw.metamodel.controlflow.impl.AltImpl") {
            combFragment.interactionOperator =
com.sodius.mdw.metamodel.uml21.InteractionOperatorKind.ALT_LITERAL;
        }
        else {
            combFragment.interactionOperator =
com.sodius.mdw.metamodel.uml21.InteractionOperatorKind.LOOP_LITERAL;
        }
        opr = target.create("InteractionOperand");
        ic = target.create("InteractionConstraint");
        oe = target.create("OpaqueExpression");

        oe.body.add(containingSection.conditionDescription);
        ic.specification = oe;
        opr.guard = ic;
        opr.name = combFragment.name;

        if (combFragment.interactionOperator ==
com.sodius.mdw.metamodel.uml21.InteractionOperatorKind.LOOP_LITERAL) {
            guardmin =
            target.create("OpaqueExpression");
            guardmax =
            target.create("OpaqueExpression");

            guardmin.body.add("1");
            guardmax.body.add("1");
            opr.guard.minint = guardmin;
            opr.guard.maxint = guardmax;
        }
    }
}

```

```

        combFragment.operand.add(opr);
        firstLineComment = target.create("Comment");
        secondLineComment = target.create("Comment");
        thirdLineComment = target.create("Comment");
        lastLineComment = target.create("Comment");
        firstLineComment.body = "0";
        secondLineComment.body = mc.isInClass + "";
        thirdLineComment.body = mc.isInMethod + "";
        lastLineComment.body = "0";
        combFragment.ownedComment.add(firstLineComment);
        combFragment.ownedComment.add(secondLineComment);

        combFragment.ownedComment.add(thirdLineComment);
        combFragment.ownedComment.add(lastLineComment);

    } if (bFound == false) {
    if (enclosingInteraction == null) {
        System.out.println("ERROR!!! enclosingInteraction == null!");
    }
    else {
        if ((fragmentqueue.fragment.last() != null)) {
            innerfrag = fragmentqueue.fragment.last();
            innerfrag.enclosingOperand =
                combFragment.operand.first();
        }
        if (bFound == false) {
            fragmentqueue.fragment.add(combFragment);
        }
    }
    containingSection = containingSection.containingSection;
}

if ((fragmentqueue.fragment.last() != null)) {
    if (enclosingInteraction == interaction) {
        enclosingInteraction.fragment.add(fragmentqueue.fragment.last());
    }
    else {
        enclosingInteraction.operand.first().fragment.add(fragmentqueue.fragment.last());
    }
}
enclosingInteraction = innermostCombFragment;
} if (target.containsSection == true) {

    @addLifelineToEnclosingOperand(sendLifeline, enclosingInteraction);
    @addLifelineToEnclosingOperand(receiveLifeline, enclosingInteraction);

    //message
    sendMessage = target.create("Message");
    sendMessage.name = ml.signature;
    methodLocation = target.create("Comment");
    methodLocation.body = mc.isInClass + " (" + mc.lineNumber.toString() + ")";
    sendMessage.ownedComment.add(methodLocation);
    methodName = target.create("Comment");
    methodName.body = mc.isInMethod;
    sendMessage.ownedComment.add(methodName);
    sendMessage.connector = connector;

    //add message to the currently open interaction fragment
    interaction.message.add(sendMessage);
    if (ml.messageSort.toString() != "reverse") {
        //Message messageSort out sort is the returned value messageSort CREATE MESSAGE LITE
        RPL,

```

```

    }
    else {
sendMessage.messageSort=com.sodius.mdw.metamodel.uml21.MessageSort.SYNCH_CALL_LITERAL;
    }

    if ( ! message )
        replyMessage = target.create("Message");
        replyMessage.name = ml.signature;
        methodLocation = target.create("Comment");
        methodLocation.body = mc.isInClass + " {" + mc.lineNumber.toString() + "}";
        replyMessage.ownedComment.add(methodLocation);
        methodName = target.create("Comment");
        methodName.body = mc.isInMethod;
        replyMessage.ownedComment.add(methodName);
        replyMessage.connector = connector;

replyMessage.messageSort=com.sodius.mdw.metamodel.uml21.MessageSort.REPLY_LITERAL;

        sendMOsS = target.create("MessageOccurrenceSpecification");
        sendMOsR = target.create("MessageOccurrenceSpecification");
        BES = target.create("BehaviorExecutionSpecification");
        replyMOsS = target.create("MessageOccurrenceSpecification");
        replyMOsR = target.create("MessageOccurrenceSpecification");

        sendLifeline.coveredBy.add(sendMOsS);
        sendMOsS.message = sendMessage;
        sendMessage.sendEvent = sendMOsS;

        receiveLifeline.coveredBy.add(sendMOsR);
        sendMOsR.message = sendMessage;
        sendMessage.receiveEvent = sendMOsR;

        BES.name = ml.signature;
        BES.start = sendMOsR;
        BES.finish = replyMOsS;

        receiveLifeline.coveredBy.add(BES);

        receiveLifeline.coveredBy.add(replyMOsS);
        replyMOsS.message = replyMessage;
        replyMessage.sendEvent = replyMOsS;

        sendLifeline.coveredBy.add(replyMOsR);
        replyMessage.receiveEvent = replyMOsR;
        replyMOsR.message = replyMessage;

        messagequeue.fragment.add(replyMessage.sendEvent);

        if (enclosingInteraction == interaction) {
            enclosingInteraction.fragment.add(sendMOsS);
            enclosingInteraction.fragment.add(sendMOsR);
            enclosingInteraction.fragment.add(BES);
        }
        else {
            enclosingInteraction.operand.first().fragment.add(sendMOsS);
            sendMOsS.covered.coveredBy.add(enclosingInteraction);

            if
(enclosingInteraction.ownedComment.get(1).body.contains(mc.isInClass) &&
enclosingInteraction.ownedComment.get(2).body.contains(mc.isInMethod)) {
@addLineNumberToAltEnclosingInteraction(enclosingInteraction, mc.lineNumber);
            }

            enclosingInteraction.operand.first().fragment.add(sendMOsR);

            sendLifeline.coveredBy.add(enclosingInteraction);
            receiveLifeline.coveredBy.add(enclosingInteraction);

```



```

        mcnew = mc.isInMethod.substring(0, 4);
        mlnew =
messagequeue.fragment.last().event.operation.name.toString().substring(0, 4);

        if ((mcnew.equals("new ") && (mlnew.equals("new "))) {
            return;
        }

        // if BEES t at return call t is VCS
        bFound = false;
        foreach (itr : uml21.BehaviorExecutionSpecification in
target.getInstances("BehaviorExecutionSpecification")) {
            if (itr.finish == messagequeue.fragment.last()) {
                enclosingInteraction = itr.owner;
                bFound = true;
                break;
            }
        }

        if (messagequeue.fragment.last().getClass().toString().equals("class
com.sodius.ndw.metamodel.uml21.internal.impl.MessageOccurrenceSpecificationImpl")) {
            interaction.message.add(messagequeue.fragment.last().message);
            // add to current interaction
            enclosingInteraction.fragment.add(interaction.message.last().sendEvent);
            // add to current interaction
            enclosingInteraction.fragment.add(interaction.message.last().receiveEvent);
        }
        else {
            enclosingInteraction.fragment.add(messagequeue.fragment.last());
        }
        @moveQueuedMessages(ml, mc, messagequeue, fragmentqueue);
    }

    private rule createInteraction::terminateQueuedMessages( messagequeue :
uml21.Interaction, fragmentqueue : uml21.Interaction) {
        enclosingInteraction = interaction;

        if (messagequeue.fragment.isEmpty() == true) {
            return;
        }

        // find BEES t at return call t is VCS
        bFound = false;
        foreach (itr : uml21.BehaviorExecutionSpecification in
target.getInstances("BehaviorExecutionSpecification")) {
            if (itr.finish == messagequeue.fragment.last()) {
                enclosingInteraction = itr.owner;
                bFound = true;
                break;
            }
        }
        if (bFound == false) {
            enclosingInteraction = interaction;
        }

        if (messagequeue.fragment.last().getClass().toString().equals("class
com.sodius.ndw.metamodel.uml21.internal.impl.MessageOccurrenceSpecificationImpl")) {
            interaction.message.add(messagequeue.fragment.last().message);

            enclosingInteraction.fragment.add(interaction.message.last().sendEvent);

            enclosingInteraction.fragment.add(interaction.message.last().receiveEvent);
        }
        else {
            enclosingInteraction.fragment.add(messagequeue.fragment.last());
        }
        @terminateQueuedMessages(messagequeue, fragmentqueue);
    }

```

```

private rule createInteraction::addLifelineToEnclosingOperand(ll :
uml21.Lifeline, intFrag : uml21.InteractionFragment) {

    while ((intFrag != null)
        &&('intFrag.getClass().toString().equals("class
com.sodius.mdw.metamodel.uml21.internal.impl.Interaction"))){
        if (intFrag.enclosingOperand != null) {
            ll.coveredBy.add(intFrag.enclosingOperand.owner);
            @addLifelineToEnclosingOperand(ll,
intFrag.enclosingOperand.owner);
        }
        intFrag = intFrag.enclosingOperand.owner;
    }

    private rule createInteraction::addLineNumberToAltEnclosingInteraction(e1 :
uml21.CombinedFragment, lineNumber : int) {
        if (e1 == null){
            return;
        }
        if (e1.interactionOperator ==
com.sodius.mdw.metamodel.uml21.InteractionOperatorKind.ALT_LITERAL){
            firstLine = Integer.parseInt(e1.ownedComment.first().body );
            lastLine = Integer.parseInt(e1.ownedComment.last().body);

            if ((firstLine == 0) || (firstLine > lineNumber)){
                e1.ownedComment.first().body = lineNumber + "";
            }
            if (lastLine < lineNumber){
                e1.ownedComment.last().body = lineNumber + "";
            }
        }
        @addLineNumberToAltEnclosingInteraction(e1.enclosingOperand.owner,
lineNumber);
    }
}

```

Figure 69 – Transformation ruleset


```

package reverseengineering;public class SignatureComparator {

    public static boolean match(String methodLogSignature, String methodCallSignature)
    {
        String packageName = methodLogSignature.substring(0,
methodLogSignature.lastIndexOf(".") + 1);
        String mlSignature = methodLogSignature.substring(0,
methodLogSignature.indexOf("("));

        if (packageName.contains("new")){
            mlSignature = mlSignature.replace(packageName, "new ");
        } else {
            mlSignature = mlSignature.replace(packageName, "");
        }

        String mcSignature = methodCallSignature.toString();
        if (methodCallSignature.indexOf("(") > 0){
            mcSignature = methodCallSignature.substring(0,
methodCallSignature.indexOf("("));
        }
        if (mlSignature.compareTo(mcSignature) == 0)
        {
            return true;
        }
        return false;
    }

    public static boolean matchQueueMessageEventOperationVsMcIsInMethod(String
queueSignature, String isInMethodSignature){
        String qSignature;
        String mSignature;
        if (queueSignature.indexOf("(") > 0) {
            qSignature = queueSignature.substring(0,
queueSignature.indexOf("("));
        }
        else {
            qSignature = queueSignature;
        }
        if (isInMethodSignature.indexOf("(") > 0) {
            mSignature = isInMethodSignature.substring(0,
isInMethodSignature.indexOf("("));
        }
        else {
            mSignature = isInMethodSignature;
        }
        if (qSignature.lastIndexOf(".") > 0) {
            qSignature = qSignature.substring(qSignature.lastIndexOf(".") + 1);
        }
        if (mSignature.lastIndexOf(".") > 0) {
            mSignature = mSignature.substring(mSignature.lastIndexOf(".") + 1);
        }
        if (qSignature.contains("new ")){
            qSignature = qSignature.replace("new ", "");
        }
        if (mSignature.contains("new ")){
            mSignature = mSignature.replace("new ", "");
        }
        if (qSignature.compareTo(mSignature) == 0)
        {
            return true;
        }
        return false;
    }

}
};

```

Figure 70 – SignatureComparator.java