USING ASPECT-ORIENTED PROGRAMMING TO INSTRUMENT OCL CONTRACTS IN JAVA

By
Wojciech Dzidek

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfilment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

December 2003

Copyright © 2003 by Wojciech Dzidek

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89833-4

Our file Notre référence

ISBN: 0-612-89833-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

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

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

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

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

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

Canada

Abstract

Analysis and design by contract allows the definitions of a formal agreement between a class and its clients, expressing each party's rights and obligations. Contracts written in the object-constraint language (OCL) are known to be useful technique to specify the precondition and postcondition of operations and class invariants, making the definition of object-oriented analysis or design elements more precise.

In this thesis, we introduce the *ocl2j* approach to automatically instrument OCL constraints in Java programs using aspect-oriented programming. The approach strives for efficient generation of contract code and a non-intrusive instrumentation technique.

Also presented is a prototype tool that provides automated support for the proposed approach. The tool is applied to a case study to verify the methodology and assess the practical challenges of its implementation.

We conclude that the approach produces positive results for most programs and we propose precise strategies to optimize instrumentation so as to further decrease overhead.

Acknowledgements

I cannot thank enough Dr. Briand and Dr. Labiche, for their mentorship and financial support throughout the course of my master's degree.

Many thanks to the Dresden OCL Toolkit team for their pioneering work in this field, the fact that they made their work open source, and for providing support for their tool.

I would like to say a special thanks to the SQUALL Research Group who made a friendly environment to work together.

Further, I would like to thank my parents for their constant support throughout my studies.

Table of Contents

ABSTRACT	III
ACKNOWLEDGEMENTS	IV
TABLE OF CONTENTS	V
LIST OF FIGURES	XI
LIST OF TABLES	XI
1 INTRODUCTION	1
1.1 THESIS CONTRIBUTION	7
1.2 THESIS ORGANIZATION	8
2 BACKGROUND INFORMATION	10
2.1 FORMAL ELEMENTS IN SOFTWARE DEVELOPMENT	10
2.2 DESIGN BY CONTRACT	12
2.3 THE OBJECT CONSTRAINT LANGUAGE	15
2.4 ASPECT-ORIENTED PROGRAMMING (AOP) TECHNOLOGY	16
2.4.1 <i>AspectJ Examples</i>	18
3 RELATED WORK	23
3.1 PARSING OF OCL EXPRESSIONS	25
3.2 SEMANTIC ANALYSIS	26
3.3 NORMALIZATION	27
3.4 CODE GENERATION	28

3.4.1	<i>Description of the OCL Type Library</i>	29
3.5	INSTRUMENTATION	32
3.5.1	<i>Pre- and Postcondition Checking</i>	32
3.5.2	<i>Invariant Checking</i>	34
3.5.3	<i>Detecting Collection Modification</i>	38
3.6	DRESDEN OCL TOOLKIT'S LIMITATIONS – A SUMMARY	40
3.7	THE TOOLKIT USED WITH ASPECTJ	44
3.7.1	<i>Invariants on Properties of One Class</i>	46
3.7.2	<i>Invariants on Properties of Associated Classes</i>	46
3.7.3	<i>Invariants on Properties of Associated Collections</i>	47
3.7.4	<i>Invariants on Properties of Elements in Collections</i>	47
3.7.5	<i>Critique of the Work</i>	48
4	THE OCL2J APPROACH.....	50
4.1	OVERVIEW	50
4.2	EXTERNALIZATION OF CONTRACTS.....	54
4.3	OCL TO CODE TRANSFORMATIONS	55
4.4	CHECKING OF CONTRACTS.....	58
4.5	INHERITANCE OF CONSTRAINTS	58
4.6	ASSUMPTIONS	59
4.6.1	<i>OCL Syntax Reduction</i>	59
4.6.2	<i>forAll Collection Operation</i>	62
4.6.3	<i>collection->sum() : T</i>	62
4.6.4	<i>collection->sortedBy(expr : OclExpression) : Sequence(T)</i>	63
4.6.5	<i>User Defined Collections Used in OCL Constraints</i>	63

4.6.6	<i>Protected & Default Access Methods</i>	63
4.7	LIMITATIONS	64
4.7.1	<i>Enumerations</i>	65
4.7.2	<i>OclState and oclInState</i>	67
4.8	SCALABILITY	67
5	THE OCL PARSER	69
5.1	SABLECC – AN OBJECT-ORIENTED PARSER GENERATOR	69
5.2	SYNTAX TREES	72
5.3	TREE-WALKERS	74
5.3.1	<i>The Visitor Pattern</i>	75
5.3.2	<i>Visitors and Tree-Walkers in SableCC</i>	78
6	THE OCL TO JAVA TRANSFORMATIONS	82
6.1	TRANSFORMATION RULES AND USAGE	82
6.2	RESOLUTIONS OF JAVA TYPES	91
6.3	OBJECT EQUALITY	95
6.4	OCL COLLECTIONS TO JAVA COLLECTIONS	95
6.4.1	<i>OCL Bag and Sequence in Java</i>	96
6.4.2	<i>Additional Collections Operations</i>	96
6.4.3	<i>Default Collection Type Equivalents in Java</i>	100
6.5	USING PREVIOUS PROPERTY VALUES IN OCL POSTCONDITIONS	101
7	OCL2J AND AOP	103
7.1	USAGE OF AOP IN THIS WORK	103
7.2	OCLTYPE : : ALLINSTANCES () AND OCLANY : : OCLISNEW ()	108

7.3	IMPACT ON PERFORMANCE DUE TO ASPECTJ	111
8	THE OCL2J TOOL ARCHITECTURE	114
8.1	THE FACADES	115
8.1.1	<i>The Model Facade</i>	116
8.1.2	<i>The Source Code Facade</i>	117
8.2	THE OCL2J PACKAGE	119
9	CASE STUDY	120
9.1	THE ROYAL AND LOYAL SYSTEM	120
9.2	RESULTS	123
9.2.1	<i>Bytecode Size</i>	124
9.2.2	<i>Execution Time</i>	126
9.2.3	<i>Memory Footprint</i>	131
9.2.4	<i>Assertion-Failure Scenarios</i>	133
9.3	CONCLUSION	135
10	POSSIBLE OPTIMIZATIONS	138
10.1	INTRODUCTION TO OCL COLLECTION-OPERATION OPTIMIZATIONS	140
10.2	MODIFYING THE CHECKING OF COLLECTION-OPERATIONS IN ASSERTIONS	142
10.3	DETECTING CHANGES	144
10.4	SIMPLE AND DERIVED COLLECTIONS	146
10.5	COMMON OPERATIONS NOT LISTING AN OCLExpression IN THE ARGUMENT LIST	147
10.6	COMMON OPERATIONS LISTING AN OCLExpression IN THE ARGUMENT LIST	151
10.7	PRELIMINARY RESULTS	156
10.8	SUPPORT FOR QUERY METHODS	159

10.9	MISCELLANEOUS OPTIMIZATIONS & OPEN ISSUES.....	161
10.9.1	<i>Class Mutability</i>	163
11	CONCLUSIONS AND FUTURE WORK	166
11.1	SUMMARY AND CONCLUSIONS.....	166
11.2	FUTURE WORK.....	170
APPENDIX A	GENERAL.....	172
A.1	TERM DEFINITIONS	172
A.2	ACRONYM DEFINITIONS	173
A.3	OCL GRAMMAR CHANGES	175
A.3.1	<i>Use of Qualifiers</i>	175
A.3.2	<i>Precedence of Relational Operators</i>	177
A.4	PERSON TAX EXAMPLE WITH OCL2J.....	177
APPENDIX B	DRESDEN OCL TOOLKIT DETAILS.....	179
B.1	SEMANTIC ANALYSIS DETAILS	179
B.1.1	<i>Type Checking</i>	179
B.1.2	<i>Invariants on the AST</i>	180
B.2	NORMALIZATION DETAILS	183
B.2.1	<i>Grammar reduction</i>	184
B.3	AN EXAMPLE OF THE SUS AFTER CONTRACT INSERTION	188
B.4	INVARIANT CHECKING PROBLEM.....	190
B.5	QUERY METHOD USAGE PROBLEM.....	191
B.6	CONSTRAINTS ON COLLECTION ELEMENTS PROBLEM.....	192
B.7	RUNTIME ERRORS THROWN BY A TOOLKIT-INSTRUMENTED PROGRAM	194

APPENDIX C	THE OCL GRAMMAR	195
C.1	SABLECC OCL GRAMMAR	195
C.2	OCL 1.3 GRAMMAR PROBLEMS	202
C.3	OCL 1.3 GRAMMAR CHANGES.....	203
APPENDIX D	THE TRANSFORMATION RULES	204
APPENDIX E	OCL2J USAGE DETAILS.....	214
E.1	THE DOCLET FAÇADE	214
APPENDIX F	ASPECTJ CODE FOR THE POSSIBLE OPTIMIZATIONS CHAPTER	217
F.1	UNOPTIMIZED SOLUTION	218
F.2	OPTIMIZED SOLUTION	219
APPENDIX G	REFERENCES.....	221

List of Figures

FIGURE 5-1: CLASSES GENERATED FOR THE PRODUCTION <code>POSTFIX_EXPRESSION</code>	73
FIGURE 5-2: CLASSES GENERATED FOR THE PRODUCTION <code>ADD_OPERATOR</code>	74
FIGURE 5-3: AN EXAMPLE OF THE VISITOR PATTERN.....	76
FIGURE 5-4: THE VISITOR PATTERN IN THE SABLECC FRAMEWORK.....	79
FIGURE 8-1: OCL2J SUBSYSTEM DEPENDENCIES	114
FIGURE 8-2: THE <code>MODELFACADE</code> INTERFACE	116
FIGURE 8-3: THE <code>REFLECTIONFACADE</code> CLASS	117
FIGURE 9-1: ROYAL AND LOYAL CLASS DIAGRAM	121
FIGURE 10-1 WORLD-PERSON CLASS DIAGRAM	141

List of Tables

TABLE 1: THE TOOLKIT'S EFFECTS ON PROJECT SOURCE CODE.....	41
TABLE 2: DEFINITIONS OF THE REGULAR EXPRESSION OPERATORS BNF IS AUGMENTED WITH IN EBNF	71
TABLE 3: TRANSFORMATION RULE TABLE.....	83
TABLE 4: OCL SYNTAX USED IN CONJUNCTION WITH SUBPHRASES.....	83
TABLE 5: THE <code>EXPRESSION</code> TRANSFORMATION.....	84
TABLE 6: THE <code>LOGICAL_EXPRESSION</code> TRANSFORMATIONS	85
TABLE 7: THE <code>PRIMARY_EXPRESSION</code> TRANSFORMATION	85
TABLE 8: THE <code>PRIMARY_EXPRESSION</code> WITH <code>TIME_EXPRESSION</code> TRANSFORMATION	86
TABLE 9: INTEGER, REAL & BOOLEAN TYPES IN OCL AND JAVA	94
TABLE 10: OCL/JAVA COLLECTION TYPE EQUIVALENCIES	100
TABLE 11: OCL2J TOOL METRICS	115

TABLE 12: ROYAL AND LOYAL CLASS DESCRIPTIONS	122
TABLE 13: ROYAL AND LOYAL PROGRAM METRICS	123
TABLE 14: CLASS SIZES IN ORIGINAL AND INSTRUMENTED VERSIONS OF THE SUS	126
TABLE 15: EXECUTION TIME COMPARISON.....	128
TABLE 16: MEMORY FOOTPRINT COMPARISON.....	132
TABLE 17: PRELIMINARY OPTIMIZATION RESULTS	159
TABLE 18: USAGE GUIDE FOR THE DOCLET FACADE	215

1 INTRODUCTION

Design by contract (DbC) is a technique developed by Bertrand Meyer [26], where the relationship between a class and its clients are viewed as a formal agreement, expressing each party's rights and obligations. These contracts are expressed with three types of predicates: class invariants, operation preconditions, and operation postconditions. An invariant is specified in the context of a class and is defined as being true before and after the execution of every public method in the class. A precondition is specified in the context of an operation of a class and is defined as being true before the execution of the constrained operation. And finally, a postcondition is also specified in the context of an operation of a class and is defined as being true after the execution of the constrained operation. The DbC approach, originally intended to be used during low level design (same abstraction level as the implementation language), was later expanded by several works (like [36]) to be used not only during low level design but also during analysis.

In the context of object-oriented (OO) development, the Unified Modeling Language (UML) is the de facto standard notation to be used during analysis and design and the definition of contracts is facilitated thanks to the Object Constraint Language (OCL) [6, 7].

OCL, a sublanguage of the UML, is a textual, formal specification language that allows for the compact notation of precise, side-effect free constraints on instances of UML models. Part of the compactness of OCL is owed to the fact that classes of UML class diagrams automatically belong to the type system that OCL constraints use. Therefore it is not necessary to have a separate type system definition. OCL is, for a specification language, very easy to learn, and is intuitive enough that many OCL constraints can be understood without any knowledge of the language, at least by those who are familiar with programming languages. Amongst other things, this language can be used to define constraints on models in the spirit of the DbC methodology. As OCL can specify constraints on models, using OCL the “Design by Contract” methodology can be naturally extended to the “Analysis & Design by Contract” methodology. In addition to being able to write constraints at the analysis stage, using OCL has many other advantages including:

- Like UML, OCL is a standard language.
- OCL is at a higher level of abstraction than OO programming languages leading to more concise and powerful contracts than if they were written in an implementation language.

The benefits of using contracts during software development are numerous, including helping achieve higher standards of correctness and robustness. In fact, this work is

highly motivated by previous work [9] where it was determined that instrumented contracts are good enough substitutes to hard-coded oracles in test drivers and that instrumented contracts can be used to significantly lower the effort of locating faults after the detection of a failure. Additionally, as Ghezzi, Mandrioli, and Jazayeri put it in [32] “When quality is pursued, productivity follows.”

Current and emerging software paradigms have recognized the usefulness of contracts. First, component and distributed object-oriented software (also known as middleware) rely heavily on contracts in order to achieve quality. In [31], the prominent book on the subject of component software by Clemens Szyperski, an entire chapter is devoted to the subject of contracts. In that chapter Szyperski observes that using a formal language to specify contracts would be ideal except for the detriment of the complexity associated with the usage of a formal language – something that OCL provides, formalism without the complexity associated with a formal language. Likewise in [30], a book discussing distributed object-oriented technologies, Emmerich discusses the benefits of contracts: “This notion [of contracts between clients and servers] is particularly useful in distributed systems where client and server objects are often developed autonomously. The object type of a server object declares the attributes and operations that it offers. Automatic validation can determine whether or not client and server implementations obey the contract that is embedded in the type definition.” Last, model driven architecture (MDA), also known as model driven development (MDD), is the next advancement in software

development technology. The authoritative book on the subject of MDA [25] notes that the combination of UML with OCL is at the moment probably the best way to develop high quality and high level models, as this results in precise, unambiguous, and consistent models that contain much information about the system to be implemented. The MDA approach is based on the idea that you can convert a high level model directly to code. Since this approach relies so strongly on OCL it is not unreasonable to assume that for the MDA framework to evolve effective code generating OCL tools are required.

Thus, these three technologies, currently at the center of software engineering research, are ideal candidates that would greatly benefit from advancements in contract technology. Yet in current practices we observe that, at the analysis and design levels, constraints are expressed in natural language, while at the implementation level, they are manually hard coded with the rest of the application code. This approach results in two major disadvantages:

- The manual translation of the constraints is error-prone, and moreover it makes maintenance of constraints difficult as two versions must be maintained: the constraints in the code and the constraints in the model.

- As experience in software engineering has shown, information in two different places deviates in meaning over time, and in this case this is aggravated by the fact that the information is in a different format. The source of this deviation can have several roots, for example updating code but not the analysis & design model.

Having discussed the advantages of OCL, it comes as a surprise that the language is hardly used at all. One reason for this is the well-established prejudices against any formal elements among software development experts and many influential methodologists. Another reason for the unsatisfactory utilization of OCL is the lack of tools. The most basic support for developers using OCL would be to check constraints for syntactic and semantic correctness. Even this is not offered by state-of-the-art CASE tools like IBM Rational Rose and Borland Together ControlCenter. It is worth noting that the OCL Compiler developed by Cybernetic Intelligence GmbH [37] adds support for syntactic and limited semantic correctness (conflicting constraints are not detected) to IBM Rational Rose. Yet even this is not enough. To gain maximum benefits from using OCL, tools must be able to generate code from OCL constraints and instrument this code into the target program's code so that the constraints can be enforced at runtime. The benefits of this kind of constraint enforcement is shown in [9], where a rigorous empirical study showed that contract assertions detected a large percentage of failures, ranging from 75% to 80%, depending on the level of precision of the contract definitions. In

addition to this, diagnosability improves nearly an order of magnitude between programs without contract assertions and the ones with contract assertions. This suggests significant savings during debugging, as faults will be much easier to locate if contracts are being instrumented and checked during execution.

Until now, the only tool that would perform this step was the code generation component of the Dresden OCL toolkit [2, 3]. Analysis of the tool quickly shows that the proposed solution is only adequate for very small and simple programs with trivial constraints. Specific problems with the tool include:

- Extremely inefficient generated code combined with a very intrusive strategy leading to a very large memory and processing overhead.
- Incompletely implemented OCL functionality (for example, `OclType::allInstances()` and `oclAny::oclIsNew()` operations are not supported).
- Worst of all, incompletely enforced constraints (for example, the validation of a constraint on elements of a collection will not be detected if the state of the collection is not modified at the same time).

1.1 Thesis Contribution

This thesis reports on the implementation of `ocl2j`, a tool which converts OCL constraints into Java code and then instrument this code in target programs. `ocl2j` was designed with completeness, efficiency, correctness, and scalability in mind. Based on the approach presented in this thesis, `ocl2j` is the first tool to achieve these criteria unlocking the full power of OCL as OCL contracts no longer have to be converted to code manually.

In summary, the main contributions of this thesis are:

- Complete¹ support for OCL 1.3.
- Transformation from OCL types and operations into Java types and operations. The transformations account for any incompatibilities between the types of the two languages.
- Efficient generation of contract code.
- Correct enforcement of the contracts.

¹ Except for supporting `oclInState(s:OclState)` and `enum{}` which was impossible to implement as explained in section 4.7. A reduced version of the grammar was implemented: human shorthand notation was not included.

- Non-intrusive instrumentation at the source code level as the instrumentation is performed at the bytecode level.
- The ability to use the tool on non-trivial programs with non-trivial constraints.
- The framework uses technology that allows for continued improvements to the tool. An entire chapter in this thesis is dedicated to this topic – Chapter 10.
- The development of a test system (used in the case study) with many precise and often complex constraints.

Finally, the work discusses possible shortcomings of the presented approach and presents, in detail, concrete ideas as to how the shortcomings can be overcome within the existing framework, allowing for further advancements of this technology.

1.2 Thesis Organization

Chapter 2 provides the required material to read this thesis: a discussion on formal elements in software development, an introduction to the Design by Contract methodology, an introduction to the Object Constraint Language, and finally an introduction to aspect-oriented programming. Chapter 3 discusses the related work. Chapter 4 gives an overview of the solution presented in this thesis. Chapter 5 describes the OCL parser needed to parse the constraints. Chapter 6 details the transformations

used to generate the Java code from parsed OCL constraints. Chapter 7 discusses how aspect oriented technology and AspectJ apply and are used in this work. Chapter 8 outlines the architecture of the tool developed in this work, ocl2j. Chapter 9 reports on the performed case study. Chapter 10 shows, in detail, how this work can be continued with the existing framework in order to overcome the mentioned limitations of this work. And finally, Chapter 11 presents our conclusions and points out directions for future work.

2 BACKGROUND INFORMATION

This chapter presents a background of material that the reader must be familiar with to fully appreciate this thesis. First, formal elements in software development are discussed with respect to the use of formal methods and languages in the area of software engineering today (in section 2.1). Next, the methodology of Analysis & Design by Contract as conceived by Bertrand Meyer is presented in section 2.2. This is followed by a brief introduction to OCL given in section 2.3. The chapter concludes with section 2.4, an introduction to aspect-oriented programming.

2.1 Formal Elements in Software Development²

The difference between “best practice” as seen by the software industry and the “state of the art” defined by researchers is well known, and is not a problem that is unique to computer sciences. Even the use of systematic, semi-formal methodologies and notations such as UML is not as wide-spread in practice as it is in lectures. The discrepancy becomes even more obvious when the formal specification of software systems through

² Parts of this section have paraphrase material found in [2].

textual languages is concerned. While large parts of the academic world of computer science put great effort into this area, it is hardly used at all in the industry. Only few projects where special emphasis is put on reliability employ some highly specialized tools and processes. This could lead to the conclusion that formal specification is not useful, because software developers would employ formal specification if it would help them. But quite clearly, the development of software could benefit from certain formal aspects that reduce the ambiguity of descriptions and allow precise modeling on a high abstraction level. This allows the detection of errors early in development hence reduces costs and increases product quality. The need for precise specifications is of special interest in the fast growing area of component software, where the behaviour of components needs to be clearly defined.

Perhaps the example of state machines is encouraging to advocates of formal specification: originally a mathematical concept, they are now for a long time an important part of accepted development methodologies. But a closer look reveals an important detail: A software developer that habitually deals with finite state machines would probably not describe them as a quintuple of a set of input characters, a set of states, a function from these two sets into the set of states, an entry state and a set of final states, while mathematicians do so without second thought. A more practical view, namely state charts, had to be developed to make the concept of state machines efficiently usable.

2.2 Design by Contract³

The notion of design by contract is central in the systematic approach to object-oriented software development, and in our opinion is as important as the rest of object technology - as important as classes, objects, inheritance, polymorphism and dynamic binding, which they complement.

A major component of quality in software is reliability: a system's ability to perform its job according to the specification (correctness) and to handle abnormal situations (robustness). Put more simply, reliability is the absence of bugs.

Reliability, although desirable in software construction regardless of the approach, is particularly important in the object-oriented method because of the special role given by the method to reusability: unless we can obtain reusable software components whose correctness we can trust much more than we trust the correctness of usual run-of-the-mill software, reusability is a losing proposition.

Research has shown that some of the most important components that help build reliable object-oriented software are: static typing, garbage collection, and reusability. But this is not enough. To be sure that object-oriented software will perform properly, a systematic

³ Parts of this introduction to Design by Contract paraphrase material found in [33].

approach is needed to specify and implement object-oriented software elements and their relationships in a software system. Design by Contract, developed by Bertrand Meyer, is such a method [26]. Under the Design by Contract theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations – contracts.

The benefits of Design by Contract include the following:

- A better understanding of the object-oriented method and, more generally, of software construction.
- A systematic approach to building bug-free object-oriented systems.
- An effective framework for debugging, testing and, more generally, quality assurance.
- A method for documenting software components.
- Better understanding and control of the inheritance mechanism.
- A technique for dealing with abnormal cases, leading to a safe and effective language construct for exception handling.

The Design by Contract methodology employs three different types of assertions to express correctness constraints for object-oriented programs. Preconditions express

properties that must be true on method entry to ensure programmatically correct execution. Postconditions express properties that must be true upon method exit, assuming that the preconditions have been met. Class invariants express properties that must be true for class objects when they are in an observable state (i.e., after object creation and before and after every feature call). In a nutshell, Design by Contract states that if the preconditions are met, the postconditions will be guaranteed and the class invariants will be preserved. As a result, Design by Contract provides a solid mathematical foundation for reasoning about the correctness of object-oriented programs.

Design by Contract also provides a powerful model for object-oriented debugging. Under this model, clients are responsible for ensuring preconditions and servers are responsible for guaranteeing postconditions and preserving invariants. Accordingly, when a precondition is violated, the bug may be attributed to client code. Similarly, when a postcondition or invariant is violated, the bug may be attributed to server code. With runtime assertion monitoring, the whole process of bug attribution becomes fully automated. In light of these substantial benefits, it is not surprising that correctness constraints rank in the top ten among the requests for enhancement to the Java Programming Language [27].

The concepts of Design by Contract are captured in the Object Constraint Language, a sublanguage of the UML modeling language in which one can write assertions about

UML models, in turn allowing for not only “Design by Contract” but also for “Analysis by Contract”. The Object Constraint Language is introduced in the next section.

2.3 The Object Constraint Language

The Object Constraint Language is a language that enables one to describe expressions and constraints on object-oriented models and other object modeling artifacts. An expression is an indication or specification of a value. A constraint is a restriction on one or more values of (part of) an object-oriented model or system.

Various constraint languages have been used in object-oriented modeling methods (Syntropy [43], Catalysis [44], and BON [45]), and programming languages (Eiffel). The OCL is a standard query language, which is part of the Unified Modeling Language set by the Object Management Group (OMG) as a standard for object-oriented analysis and design.

OCL is a very important addition to the standard language of OO modeling as modelers use it to express those additional nuances of meaning which the diagrams alone cannot represent. In other words, OCL is used to give added precision to the definition of UML.

The Object Constraint Language (OCL), developed by IBM, is part of the Unified Modeling Language (UML) from version 1.1 on. This extension has been designed to

augment a class diagram with additional information which cannot be otherwise expressed by UML diagrams; previous versions of UML have only allowed the definition of constraints as annotations in an informal textual way. OCL allows the definition of integrity constraints at the user level, and it has also been used for the formalization of the metamodel of UML. The introduction of a constraint language is an important step towards the formalization of system specification. Constraints represent necessary conditions for a domain to constitute a model of the static aspects of the specified system. OCL is based on standard set theory and it was designed to specify invariants on classes and types in the class model, to specify type invariant for stereotypes, to describe pre- and postconditions on operations and methods, to describe guards, and it is also suited to specify queries in the database sense. That is, OCL can be used to write expressions that evaluate to “true” or “false” and also to write expressions that once evaluated return the values respectively satisfying the constraint specified by those query expressions.

2.4 Aspect-Oriented Programming (AOP) Technology

Aspect-oriented programming is a new technology for dealing explicitly with separation of concerns in software development. In particular, it deals with those concerns that crosscut the modularity of traditional programming mechanisms. For example, code that implements a particular security policy would have to be distributed across a set of

classes and methods that are responsible for enforcing the policy. However, with aspect-oriented technology, the code implementing the security policy could be factored out from all the classes to an aspect. Thus, the aspect localizes in one cohesive place the code that affects the implementation of multiple classes and methods [18, 19].

Aspects make it possible to create cohesive modules that implement specific concerns (like logging information – the executing method’s name and actual parameters) that otherwise would have to be distributed across many primary abstractions (for example, the primary functionality of an operation). By placing these concerns separately in an aspect, the primary abstractions are made more cohesive since their implementations are relieved of the burden of managing concepts unrelated to their purpose. Later, this code would be integrated with the classes that must support this policy by a process known as weaving. Weaving injects the code of an aspect into well-defined locations, called joinpoints, in the syntactic structure of a primary abstraction.

The practical consequence of writing aspects is that less code is written. All the code that would otherwise be distributed throughout a collection of primary abstractions is now localized, thus reducing redundancies. A key observation here is that the originally distributed code actually has a modular structure of its own, and this is the key idea behind aspect-oriented programming [18].

This work uses AspectJ (AOP for Java) – the most successful implementation of AOP

today. Next, examples of AspectJ usage are shown and explained.

2.4.1 AspectJ Examples⁴

Weaving Example

The weaver, a processor, assembles an individual concern in a process known as weaving. The weaver, in other words, interlaces different execution-logic fragments according to some criteria supplied to it. To illustrate code weaving let's consider a very basic credit card processing system example. For brevity, consider only two operations: credit and debit. Also assume that a suitable logger is available. Consider the following credit card processing module:

```
public class CreditCardProcessor {
    public void debit(CreditCard card, Currency amount)
        throws InvalidCardException, NotEnoughAmountException,
               CardExpiredException {
        // Debiting logic
    }

    public void credit(CreditCard card, Currency amount)
        throws InvalidCardException {
        // Crediting logic
    }
}
```

Also, consider the following logging interface:

⁴ Some material in this chapter has been paraphrased from [20].

```
public interface Logger {
    public void log(String message);
}
```

The desired composition requires the following weaving rules, expressed here in natural language (a programming language version of these weaving rules follows):

- Log each public operation's beginning
- Log each public operation's completion
- Log any exception thrown by each public operation

The weaver would then use these weaving rules and concern implementations to produce the equivalent of the following composed code:

```
public class CreditCardProcessorWithLogging {
    Logger _logger;

    public void debit(CreditCard card, Money amount)
        throws InvalidCardException, NotEnoughAmountException,
               CardExpiredException {
        _logger.log("Starting CreditCardProcessor.credit(
                    CreditCard, Money) " + "Card: " + card + " Amount: "
                    + amount);

        // Debiting logic
        _logger.log("Completing CreditCardProcessor.credit(
                    CreditCard, Money) " + "Card: " + card + " Amount: "
                    + amount);
    }

    public void credit(CreditCard card, Money amount)
        throws InvalidCardException {

        System.out.println("Debiting");
        _logger.log("Starting CreditCardProcessor.debit(
                    CreditCard, Money) "
                    + "Card: " + card + " Amount: " + amount);
    }
}
```

```

        // Crediting logic
        _logger.log("Completing CreditCardProcessor.credit(
            CreditCard, Money) "+ "Card: " + card + " Amount: "
            + amount);
    }
}

```

Below is an AspectJ implementation of the logging aspect for the weaver previously described in natural language. The main point to notice is that the credit card processing itself does not know anything about logging:

```

public aspect LogCreditCardProcessorOperations {
    Logger logger = new StdoutLogger();

    pointcut publicOperation():
        execution(public * CreditCardProcessor.*(..));

    pointcut publicOperationCardAmountArgs(CreditCard card,
        Money amount):
        publicOperation() && args(card, amount);

    before(CreditCard card, Money amount):
        publicOperationCardAmountArgs(card, amount) {
            logOperation("Starting",
                thisjoinpoint.getSignature().toString(), card, amount);
        }
    after(CreditCard card, Money amount) returning:
        publicOperationCardAmountArgs(card, amount) {
            logOperation("Completing",
                thisjoinpoint.getSignature().toString(), card, amount);
        }
    after (CreditCard card, Money amount) throwing (Exception e):
        publicOperationCardAmountArgs(card, amount) {
            logOperation("Exception " + e,
                thisjoinpoint.getSignature().toString(), card, amount);
        }
    private void logOperation(String status, String operation,
        CreditCard card, Money amount) {
        logger.log(status + " " + operation +
            " Card: " + card + " Amount: " + amount);
    }
}

```

HelloWorld with AspectJ

To keep with the tradition of introducing a new programming language with a “Hello World” program, the following is one in AspectJ.

First, here's a simple class containing methods to print a message:

```
// HelloWorld.java
public class HelloWorld {
    public static void say(String message) {
        System.out.println(message);
    }

    public static void sayToPerson(String message, String name) {
        System.out.println(name + ", " + message);
    }
}
```

The following aspect is the implementation for adding greeting and gratitude manners. Before the program prints a message, it should print "Good day!", and should follow the message with "Thank you!":

```
// MannersAspect.java
public aspect MannersAspect {
    // say*(..) means any method that starts with say and regardless of
    // the number of parameters it has.
    pointcut callSayMessage() : call(public static void
                                   HelloWorld.say*(..));
    before() : callSayMessage() {
        System.out.println("Good day!");
    }

    after() : callSayMessage() {
        System.out.println("Thank you!");
    }
}
```

The MannersAspect.java file, in a process similar to class declaration in Java, declares a

MannersAspect aspect. The aspect defines a `callSayMessage()` pointcut that captures calls to all public static methods with names that start with `say`. In this example, it would capture `say()` and `sayToPerson()` in a `HelloWorld` class taking any arguments. Then you define two advices for before and after reaching the `callSayMessage()` pointcut: printing "Good day!" and "Thank you!" When you compile the program with the AspectJ compiler, then run it, you'll see "Good day!" printed before each message, and "Thank you!" after each message.

For a quick reference to the AspectJ language please see [21].

3 RELATED WORK

This chapter takes a look at the state of the art in the field of OCL to Java code generation and instrumentation. It is observed that only two academic efforts exist in this area with only one of those solutions being available to the general public while the second solution is based on the first. No commercial solutions are available.

Currently, the only tool that can take an OCL 1.3 (Object Constraint Language) expression and convert it to Java code is the Dresden OCL toolkit (referred to as the toolkit from this point on) [2] and [3]. This work will now be examined in detail outlining their methodology and then discussing the limitations of their work.

Dresden OCL, developed at the University of Dresden, is based on Frank Finger's master's thesis (spring of 2000) [2]. The original version consists of an OCL compiler that compiled OCL expressions into Java expressions. Later, the tool has been expanded with instrumentation so that the generated Java code could be inserted into the source code [3]. Frank Finger's work is based on several earlier works with the most significant

being the implementation of an OCL type library⁵ [5] that implements the OCL type-system and is used in the generated Java code.

Below is a short description of how the toolkit works with the details of each step discussed in subsequent sections.

1. OCL expressions supplied by the XMI (XML Metadata Interchange) model or the comments in the source code are parsed using a LALR(1) parser generated with the SableCC parser generator. The parsing of OCL expressions is described in section 3.1.
2. A semantics analysis (discussed in section 3.2) is performed to uncover errors not detected by the parser.
3. Subsequently, the OCL expression is simplified to make the code generation easier. This is called normalization and is discussed in section 3.3.
4. The code is generated using a model of system under study (SUS). Code generation is discussed in section 3.4.
5. The generated code is inserted into the source code; this process is called instrumentation and is discussed in section 3.5.

⁵ This is known as the OCL Basis Library in the Dresden OCL Toolkit.

The work is then summarized and critiqued in section 3.6. Section 3.7 discusses the work presented in [22], where the work from [2] is used in conjunction with AspectJ.

3.1 Parsing Of OCL Expressions

When parsing OCL expressions the first step is to transform the expression into a syntax tree (performed by a parser). Other modules, like a type checker or a source code generator, take the syntax tree generated by the parser as their input.

For the OCL Parser, the tool SableCC [4] was chosen. SableCC was chosen due to its ability to not have to mix the grammar specification with action code, instead it produces a compiler framework with a parser and tree walkers. The generated parser produces an abstract syntax tree (AST). The classes whose objects constitute the abstract syntax tree are themselves generated automatically from the grammar specification. The tree walkers traverse the abstract syntax tree and can be adapted to handle different tasks through subclassing.

This property of SableCC is advantageous in their work since several modules need to perform different tasks with the abstract syntax tree. This is easily achievable by implementing modules as independent AST passes.

The grammar used to generate the parser is a rewritten version of the grammar from the

OCL specification, modified to fit the syntax understood by SableCC⁶. For the (minor) discrepancies between the official OCL grammar and the one used by the toolkit please consult appendix A.3. The main task of the parser is to split the OCL contract into tokens (lexical analysis) and then to build an AST from these tokens. The nodes of this tree are objects of classes that are automatically generated by SableCC from the OCL grammar.

3.2 Semantic Analysis

As the parser will not detect semantically erroneous OCL expressions, special consideration must be given to semantic errors so as to increase the usability of the tool.

One example of these errors is type errors (the type used does not conform with an expected type, for example if an operand of the operator '+' is not conformant to Real), other consistency rules can be expressed as constraints on the syntax tree (see appendix B.1.2). Additionally, the type checking makes type information available for later compiler phases. These checks should take place before normalization and code generation so that the implementation of these compiler modules is simplified.

⁶ Since the grammar as defined in [6] is not LALR(1)-parsable and SableCC supports no other parsing algorithm than LALR(1), the grammar had to be changed.

Since the work presented in this paper assumes correct OCL expressions (both, semantically and syntactically) the details of the semantic analysis subsystem are not directly relevant, but can be found in appendix B.1.

3.3 Normalization

During the normalization the original AST is transformed into a new AST which does not use the entire OCL grammar (grammar reduction). Normalization is performed to make the implementation of the code generation easier. Generally speaking, a syntax tree is considered simpler than a semantically equivalent syntax tree if it conforms to a grammar with fewer alternatives (an example of an alternative for the mathematical operation $+$ is a double $-$, $+1 = -(-1)$, etc.) or to a type system with fewer type features (the operations a type supports, for example an Integer supports $+$, $-$, etc.). A simplified AST will usually be bigger, i.e. contain more nodes, than the original tree.

This work uses a subset of the OCL syntax that can be trivially derived from the full OCL syntax. Thus, the details of the normalization subsystem found in the toolkit are not directly relevant to this work, but are described in appendix B.2.

3.4 Code Generation

The code generator traverses the normalized AST and builds expressions in the target language for the target system. The generated code is constructed in such a way that it uses reflection at runtime to determine the implementation details (e.g., How is the collection implemented in the SUS?). All Java variables used in the contracts are wrapped by the equivalent OCL types provided by the toolkit, the OCL Type Library; hence the generated Java code consists mostly of calls to the OCL type library. For example, a Java `int` primitive and `Integer` object will both be wrapped by `tudresden.oc1.lib.OclInteger`. This method is what we would call a “bottom-up” approach, i.e. from Java code to OCL. First, all variables used in the contract must be converted to equivalent OCL types found in the OCL type library, only then can contract checking be performed. Using the wrapper provides two advantages: the generation of the assertion code is straightforward and static analysis of the code is not necessary to determine the Java types for variables used in contracts (the distinction between a collection being a `Set` and a `Sequence`). This is further discussed and described in subsequent sections.

3.4.1 Description of the OCL Type Library

The OCL type system as defined in the UML 1.3 specification is represented in the toolkit with a library containing classes for each OCL type. There are a few reasons to use these custom types rather than the similar Java types that exist for most of these OCL types. One is that many of the types and operators in OCL works in a slightly different way than in Java. For example the operator `=` is legal for comparing an Integer and an Object in OCL, but not in Java. The collection types are also different; OCL 1.3 does not support nested collections (collections where elements are other collections), but Java does. Another reason is that some of the OCL types and collections need support in the type classes for special OCL operations, such as `forAll` and `allInstances`.

In addition to the type classes, the library contains functionality to get value and type information from attributes of the SUS. This functionality uses the Java reflection API. If a postcondition contains references to the `@pre` value of variables, code is generated to save this value during the method call.

The advantage of using such a library is simplification:

- The translation of the contract to code is trivial as the OCL wrappers provide all the methods of the OCL types.

- Static analysis of the code is not necessary to determine the Java types for variables used in contracts (as this is done dynamically by the wrapper). This information is necessary as a collection in OCL can be implemented in Java in many different ways, thus operations on the OCL type map to different methods depending on which class the collection is implemented with. For example, from OCL's perspective both, a class implementing the `java.util.Collection` interface and a reference can be treated as collections (collection-operations can be used upon either).

The drawback of using such a library is that every variable, collection, and constant used in the contract is wrapped by a type of this library resulting in a large memory overhead and big performance penalty.

Examples

The following example displays how the code generator in the toolkit works. The contract:

```
context Company inv: manager.age >= 18
```

is represented by this Java fragment:

```
final OclAnyImpl tuddOclNode0 = Ocl.toOclAnyImpl(Ocl.getFor(this));
final OclAnyImpl tuddOclNode1 =
    Ocl.toOclAnyImpl(tuddOclNode0.getFeature("manager"));
final OclInteger tuddOclNode2 =
    Ocl.toOclInteger(tuddOclNode1.getFeature("age"));
final OclInteger tuddOclNode3 = new OclInteger(18);
final OclBoolean tuddOclNode4 = tuddOclNode2.isGreaterEqual(tuddOclNode3);
```

The first statement uses reflection to get an `OclAnyImpl` object of the current class: `Company`. The second statement uses reflection's feature *manager*. The third statement uses reflection to get the value of the *age* field of the *manager* object. The fourth statement creates an `OclInteger` with the value set to 18. Finally all the necessary objects are created and the test can be performed in the fifth element. The variables are prefixed with `final` so that this dynamic retrieval of the Java types is performed only once.

Next, this contract:

```
context Klassen inv: attribute_1=0
```

generated:

```
final OclAnyImpl tudOclNode0 = Ocl.toOclAnyImpl(Ocl.getFor(this));
final OclInteger tudOclNode1 =
    Ocl.toOclInteger(tudOclNode0.getFeature("attribute_1"));
final OclInteger tudOclNode2 = new OclInteger(0);
final OclBoolean tudOclNode3 = tudOclNode1.isEqualTo(tudOclNode2);
```

As in the first example, first all variables and classes used in the contract are converted to variables of the OCL type library, then the test is executed and the result is saved in another node of type `OclBoolean`. The nodes are not saved for later use, but are generated again every time the test is run.

3.5 Instrumentation

To test contracts at runtime, the Java code generated from the OCL expressions has to be included into the program. The toolkit generates three types of code fragments (discussed later) which are inserted into the SUS in the following manner:

```
public class SomeClass {
    // The original method is renamed.
    public returnType someMethod_original() {
        // Original method code.
    }

    // Wrapper around the original method.
    public returnType someMethod() {
        // TRANSFER FRAGMENT
        {
            // PREPARATION FRAGMENT
        }

        // Invocation of the original method.
        {
            // POST FRAGMENT
        }

        return result;
    }
}
```

For a complete example of the SUS code after contract insertion please see appendix B.3.

3.5.1 Pre- and Postcondition Checking

Preconditions are checked right before the first statement in the method and postconditions are checked right after the last statement in the method. If a postcondition has @pre values, those are saved before the method executes.

Consider the following contract:

```
context Person::getIncomeAfterTax(tax:Real):Real
    post: age = age@pre
```

The following transfer fragment would have to be created. In this case the transfer node would be an OCL integer that can hold the temporary value of age@pre.

```
OclInteger tudOclNode2;
```

Next, the transfer node must be updated in the preparation fragment (this code is executed right before the invocation of the original method):

```
final OclAnyImpl tudOclNode0=Ocl.toOclAnyImpl( Ocl.getFor(this) );
final OclReal tudOclOpPar0=Ocl.toOclReal( Ocl.getFor(tax) );
final OclReal tudOclResult0=OclReal.UNDEFINED;
final OclInteger tudOclNode1=Ocl.toOclInteger(
    tudOclNode0.getFeature("age")
);
tudOclNode2=Ocl.toOclInteger(tudOclNode0.getFeature("age"));
final OclBoolean tudOclNode3=tudOclNode1.isEqualTo(tudOclNode2);
```

Comments: Only the code in bold is necessary. It is interesting to note that a lot of unnecessary code is generated. The unnecessary code would be fully used if the postcondition used the method's parameter and return value. This is not the case in our postcondition yet the supporting code is generated all the same.

Finally, the post condition is evaluated in the post fragment (right after the return from the original method):

```
final OclAnyImpl tudOclNode0=Ocl.toOclAnyImpl( Ocl.getFor(this) );  
final OclReal tudOclOpPar0=Ocl.toOclReal( Ocl.getFor(tax) );  
final OclReal tudOclResult0=Ocl.toOclReal( Ocl.getFor(result) );  
final OclInteger tudOclNode1=Ocl.toOclInteger(  
    tudOclNode0.getFeature("age")  
);  
final OclBoolean tudOclNode3=tudOclNode1.isEqualTo(tudOclNode2);
```

Comments: Again, only the code in bold is necessary. tudOclNode1 is redefined as it's already been defined in the preparation fragment.

It is easy to see how much code is needed for a simple postcondition when comparing to the constraint checking and enforcement code generated by ocl2j is found appendix A.4.

3.5.2 Invariant Checking

There are two ways to check invariants:

- Before every public method entry and after every public method exit. This is simple but inefficient as only a percentage of methods in the class may affect the invariant. Thus, the invariant may be often checked unnecessarily.
- Only after a method modifies an attribute that is also checked in an invariant. This approach is very efficient but it also not trivial to implement. The toolkit attempted to

implement this approach.

For the invariant checking system to work the following framework is needed:

- The introduction of a backup attribute for every attribute in every class. This is used to determine whether the invariant should be checked.
- The introduction of functionality to detect attribute inequality (via the `==` operator, this is discussed later in this section).
- The introduction of a collection (a `java.util.HashSet`) for every attribute in the class. This is a collection of all observers of that attribute that need to be notified if this attribute's value changes. The observers are the invariants that use the attribute (see next point).
- The introduction of an invariant attribute (of type `tudresden.ocl.injection.ocl.lib.Invariant`) in every class with an invariant.
- The introduction of functionality to check the invariant in classes with invariants.

It is important to note that the introduction of a backup attribute in every class virtually clones each object. It's also important to emphasize that this is done for every attribute in every class in the system, regardless if the attribute is used in an invariant or not. As will be explained, due to the way the invariant checking system detects attribute changes this

framework must be blindly inserted in every class, thus making the system inefficient.

The functionality to detect attribute inequality is done via the Java `==` operator (collections are dealt with in a special manner as discussed in the next section). At first glance it might seem that this system can only detect changes to attributes of primitive type and reference changes. This is not so as each attribute in every class is backed up and may be observed for change by any invariant as explained in the following example. Consider the following class diagram:

And the following invariant: context A my_inv: b.x > 0

The toolkit will add the following attributes to class A: b_backup:B, my_inv: tudresden.ocl.injection.ocl.lib.Invariant. Next, the following attributes are introduced to class B: x_backup:int, observersOf_x:HashSet. Before or after a method in A executes the invariant will be checked if:

- The value of b differs from the value of b_backup.
- my_inv was notified (via the observer pattern) that the value of x in b was changed with a method in class B.

Unfortunately, this functionality is very limited. If the value of x in b is modified via a method in class A in such a way that it falsifies the invariant then this falsification is undetected. This erroneous behaviour is exemplified with the code listed in appendix B.4.

This system suffers from severe limitations since a contract which uses the attribute of a class without source cannot be specified as the toolkit cannot insert the parts of the framework into that class. Thus, classes from third-party libraries for which the source code is not available cannot be used in contracts either directly or indirectly.

Additionally, the implementation is inefficient as objects are notified about a non-primitive type attribute change (that an invariant of that object uses) with the observer pattern. After notifying the object, the object is deregistered from the notification list. If that object is still interested in getting change notifications it must reregister with the attribute. This constant registration and deregistration quickly compounds to a large performance penalty.

The toolkit documentation specifies that it does not support query methods as discussed in section 7.3.0.0.2 of [3] “The total approach to caching requires a significant memory overhead. Observing queries will make this even worse by introducing another step of indirection. This should not be attempted until this approach has shown to be functional for big projects with many and complex invariants.” The Dresden group claims that query methods are now supported, yet a simple test program used (see appendix B.5) shows that

this is not implemented correctly as in the program a simple invariant using a query method is not enforced.

3.5.3 Detecting Collection Modification

When an attribute is a collection special consideration must be given as determining equality between collections is not as trivial as detecting equality between primitives (via the Java '==' operator). The authors correctly observe that the backup of a collection should not be a collection as well since this would consume large amounts of memory, as well as the fact that comparison between two collections is very slow. Thus, the backup for collection attributes stores a single integer value only. This value is computed from the collection. This value changes when the collection changes. The problem of detecting changes to collections (a collection has not changed if it has the same elements value wise) is addressed in three ways, giving the user a choice of the mechanism to use to monitor collections.

- **Size.** Only the number of elements in the collection is monitored. This is quick but highly inexact. If one element in the collection is replaced with a completely different element this is not detected. No changes to elements are detected, for example, the operation `forall(expression)` can be valid at one point in time but then if an element's in the collection attribute is modified that the expression depends on the constraint on the collection fails without being detected.

- A hash code. The value is derived from the hash value of every element in the collection. This is better than just looking at the size of the collection but it's very inefficient and is still inexact as `hashCode()` is not forced to return different values for different objects. Java documentation [13] for the `hashCode()` method specifies that "It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results."
- The last solution depends on the collection iterator's fail-fast methods (see the API document for `HashSet` in [13] for more information on fail-fast functionality). These methods throw an exception when they detect that there's a modification to the collection (as it's being iterated over). This solution has two problems, the first problem is, as stated in [13]: "it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs*". The second problem is that changes to elements are not detected.

Thus, no acceptable solution has been found for this problem by the toolkit authors.

The lack of enforcement of constraints on elements in the collection is exemplified by the program listed in appendix B.6.

3.6 Dresden OCL Toolkit's Limitations – A Summary

Due to the tool's "bottom-up" approach, i.e. the conversion from Java types to their own OCL-like types, the tool pays a large memory and performance penalty.

The implementation of contract checking is inefficient, incomplete, and erroneous. The inefficiency is due to all objects being virtually cloned (see section 3.5.2). The OCL operations `type.allInstances:Set(type)` and `object.oclIsNew():Boolean` are not supported, making the implementation incomplete. Finally, the implementation is erroneous as query methods usage in contracts is not properly supported (see section 3.5.2), invariants are not always enforced (see section 3.5.2), constraints on elements in collections are not properly enforced (see section 3.5.3), no support for static class attributes accessed from within constraints, class operation names cannot have the same names as OCL collection operations names, and sometimes after source code is instrumented it compiles but then runtime exceptions is thrown when the instrumented program is executed.

The toolkit's Java code instrumentator does the virtual-cloning in the name of performance, yet no performance measures are given between the optimized and an un-optimized solution. Thus, one cannot assume that these optimizations are actually beneficial.

The instrumentation of the contract checking system is very intrusive as it heavily modifies the source code – see section 3.5. The effects of the toolkit on a real world project can be seen in this table taken from [3]:

	Original Code	Instrumented Code
Size of Source Code	1.1 MB	3.2 MB
Lines of Code	17131	43587
(Re-) Instrumentation	36 s	91 s
(Re-) Cleaning	23 s	73 s

Table 1: The toolkit’s effects on project source code

Unfortunately, Table 1 is hard to interpret not knowing the number of constraints specified on the SUS nor the type of the constraints specified. After contacting the author of this work regarding this issue he said that “only a few constraints” were used with this application.

As can be seen from this table the size of the source code almost triples, this has important side effects:

- Compilation times are longer.
- A large number of additional lines of code translate to a slower execution rate.
- A large number of additional lines of code translate to larger source and bytecode files.

Additionally, since the toolkit inserts the contract code directly into the source code the

user is faced with a dilemma: keep only one version of the source code (either the clean version or the instrumented version) or keep both versions of the source code. Both options have disadvantages, when only one version of the code is kept then the user must deal with the long wait times for the cleaning and instrumentation whenever a change to the source needs to be made. If the version of the code being kept is the instrumented version then the code must be cleaned whenever the user wants to read the code. Considering the fact that on a program with 17K lines of code (LOC) it takes almost two minutes to clean and instrument the program this is a significant burden. This is especially true when considering the style of code development today where the programmer makes a small change and then runs the program to see if the change had the desired effect. Thus it is easy to see that this lag time is unacceptable during development which means that, in the real world, the toolkit cannot be used all the time during program development reducing the value of the tool.

Keeping both versions of the source code solves some of the above outlined problems but introduces new ones. If a version management system is used, two versions of the source code must be kept in the system. The first issue with this is that size of the code databases instantly more than doubles. Secondly, whenever two versions of the same thing exist an inevitable thing happens: inconsistencies are introduced amongst the two versions of the “same” code. Due to the fact that it takes a lot of time to clean and instrument the code a programmer nearing a deadline will be tempted to just change the instrumented version

of the code (this is possible but is not elegant) thinking that he or she will update the un-instrumented version of the code at a later time. If something more important comes up the programmer may forget to update the other version of the code introducing an inconsistency (between the instrumented version of the code and the un-instrumented version).

The toolkit instrumentation technique also prevents them from providing a solution for the `OclType::allInstances()` and the `oclAny::oclIsNew()` operations. `OclType::allInstances()` can be used on any type in the model (including user defined types) and it returns the set of all instances of type and all its subtypes in existence at the snapshot at the time that the expression is evaluated. `oclAny::oclIsNew()` can only be used in a postcondition and evaluates to true if the object is created during performing the operation, i.e. it didn't exist at precondition time.

A study of this tool performed by [1] determined that more work needs to be done to provide usable code generation and instrumentation, with emphasis on efficiency. The toolkit is a start, but is both incomplete and inefficient making it unusable for even slightly non-trivial programs. This dictates the need to develop a system which would solve the current limitations so that research in OCL as a design by contract language may progress.

3.7 The Toolkit Used With AspectJ

The work presented in [22] uses the toolkit's Java code generator [2] in conjunction with AspectJ to implement a contract checking system. Please note that the work in [3], which as previously explained is an instrumentation system, is not used. AspectJ is used as the instrumentation technology. This work discusses a tool that generates aspect templates using path analysis of the OCL invariant and coding conventions. The paper focuses on determining the scope of constraints, i.e. the points in the application code where the constraint should be checked (referred to as insertion points in their work). The scope of a constraint depends on the navigation path of its corresponding OCL expression. As the navigation path becomes longer, more classes become involved and more insertion points have to be specified. These insertion points are scattered over the several classes that can be distilled out of the navigation paths of the class invariant. The complexity of this issue led them to create a classification of constraints based on their navigation paths, or the classes they involve. The classification is used to determine the insertion points for the constraint checking code. The problem of pre- and postconditions is not addressed in their work as the scope of those kinds of constraints is straightforward.

The invariants are classified into four types:

1. Invariants on properties of one class.

2. Invariants on properties of an associated class.
3. Invariants on properties of associated collections.
4. Invariants on properties of elements in collections.

For each type of invariant a pattern has been developed specifying the different insertion points. Insertion points specify where and how an invariant should be checked. The paper shows how Aspect Oriented Programming (AOP) can be used to implement the checking of the different types of invariants. AspectJ is used as the AOP language.

The developed system relies on the following coding conventions:

- Only getter and setter methods modify class attributes.
- Specialized methods are used to add and delete elements to and from a collection. i.e. A class with a collection of `Person` elements would also need the following methods:
`addPerson(Person p)` to add elements to that collection and `deletePerson(Person p)` to delete elements from that collection. Thus, only those two methods can add and remove elements to and from that collection. Otherwise the constraint will not be enforced.

The insertion points given in the next sections use the OCL constraint and the UML model to determine the context class, the involved associations, attributes, operations,

and associated classes.

3.7.1 Invariants on Properties of One Class

The insertion points for this kind of invariant are: after the constructors of the class, before and after all public methods of the context class that access the involved instance variables, before and after all public methods of the context class that change the involved properties. If the involved properties of the class are available to all classes to which the context class is visible, there must be a check before and after all public methods of these classes that access or change these properties.

This unclear explanation is taken directly from [22], and the same is done for the explanation of the remaining constraint types. In the “Critique of the Work” section (3.7.5) of this chapter one of these insertion-points strategies is analysed.

3.7.2 Invariants on Properties of Associated Classes

This kind of invariant is an extension of the first type. The properties participating in the invariant do not have to belong to the same class but a combination of properties of different classes is possible.

The insertion points for this kind of invariant are: after the constructors of the context

class, before and after every public method of the context class and the classes involved in the associations that access or change the involved associations or the involved attributes. Before and after every public method of the classes to which the involved associations and attributes are visible are also insertion points.

3.7.3 Invariants on Properties of Associated Collections

The insertion points for this kind of invariant are: after the constructors of the context class, before and after every public method of the context class and the classes involved in the associations that access or change the involved associations. Also, before and after every public method of the classes to which the involved associations are visible are also insertion points.

The paper notes that it is possible to limit the number of insertion points by looking at the functionality of the OCL operations on the collections.

3.7.4 Invariants on Properties of Elements in Collections

The insertion points for this kind of invariant are: after the constructors of the context class, before and after every public method of the context class and the classes that access or change the involved associations or the involved properties, before and after every public method of the classes to which the involved associations or properties are visible.

3.7.5 Critique of the Work

Like us, the authors of [22] noticed that AOP can be for non-intrusive contract instrumentation. Unfortunately this work presents a vague and unrealistic framework for reasons that will now be explained. To start, the framework forces users to use inflexible coding conventions. A tool should not place such constraints on the user; it should rather integrate seamlessly with their coding practices and standards.

The defined insertion points are described vaguely and in a confusing manner, to prove this point the discussion on insertion points for “Invariants on Properties of Elements in Collections” will now be analyzed:

1. After the constructors of the context class.
2. Before and after every public method of the context class.

The first two points of the discussion are explained in clear manner.

3. Before and after every public method of a class that accesses or changes the involved associations or the involved properties.

This is very vague. How are these classes determined?

4. Before and after every public method of the classes to which the involved associations or properties are visible.

Again, this is very general. This could mean before and after every method in every class in the system! How are these classes determined? How is the number of these classes minimized? Is any static analysis of the code performed? None of these questions are answered in a precise and convincing manner.

The authors also observed that certain collection operations may require fewer insertion points to check their validity, but this is only mentioned in passing and not explored in any detail.

Also, because this framework is built on top of [2], it suffers from the same problems:

- A large memory and performance penalty due to the “bottom-up” approach of contract related code generation.
- No support for query methods.

4 THE OCL2J APPROACH

This chapter discusses the approach taken in this thesis, to make automatic generation and instrumentation of OCL constraints in Java possible. The chapter begins with section 4.1 where a general overview to the ocl2j (OCL to Java) approach is given. Section 4.2 describes how the assertion code is kept separate from the SUS source code. Section 4.3 discusses the OCL to Java transformations and how these transformations are superior to the ones in the toolkit. Section 4.4 discusses the topic of contract checking, i.e. when the contracts should be checked during the execution of the SUS. Section 4.5 discusses the concept of inheritance with respect to contracts. Section 4.6 discusses the assumptions made by the ocl2j approach. Section 4.7 discusses the limitations of the approach and finally section 4.8 discusses the scalability of the approach.

4.1 Overview

The section begins with an explanation as to how this approach is different from the one taken by the toolkit. Then the steps in the approach are outlined.

The ocl2j approach differs from the one taken in the toolkit as it is complete, efficient,

non-intrusive, and it ensures that constraints are properly enforced. This is accomplished by addressing the shortcomings of the approach taken in the toolkit (discussed in detail in section 3.6):

- The tool is correct in that constraints are properly enforced and none of the faults found in the toolkit are present in ocl2j.
- The instrumentation by the tool is non-intrusive, as aspect oriented programming technology is used to externalize the contracts: separation of the contracts from the program source code. Having all contract code in separate files has many advantages: having only one version of the source code eases the task of configuration management; contract checking can be enabled during development as incremental instrumentation allows for very quick weaving; no need to regenerate the assertion code if the parts of the class diagram pertinent to the constraints do not change. This is discussed in section 4.2.
- The tool is efficient as instead of converting all Java variables (participating in contracts) to their OCL equivalent variables (as is done in the toolkit - see section 3.4), a transformation system between the OCL variable types and the standard Java variable types is used during constraint code generation (outlined in section 4.3). In other words, the transformation between the OCL types and the Java types is performed statically, as opposed to the toolkit where it's done dynamically. Thus, the

ocl2j instrumented programs suffer no overhead due to wrapping. Additionally, the ocl2j solution does not “clone” all objects (as is in the toolkit – see section 3.5.2) making the amount of overhead memory caused by ocl2j much smaller.

- The tool is nearly complete⁷ as all OCL functionality is implemented (from a semantic point of view); sections 4.6 and 4.7 discuss the parts that were not implemented.

The steps involved in the ocl2j approach can be summarized as follows:

1. The model is analyzed so that the following data can be extracted: the constraints; the elements types in collections; the key types and values types in maps.

This information can be extracted from an XMI file or the SUS source code if it is enriched with the relevant metadata (e.g., the constraints). Section 8.1.1 discusses how this information is extracted within the ocl2j framework.

2. Then, for each constraint Java code is produced as follows:

2.1. The constraint is parsed and an abstract syntax tree (AST) of the constraint is created. Details on this can be found in Chapter 5.

⁷ Except for supporting for `oclInState(s:OclState)` and `enum{}` which was impossible to implement as explained in section 4.7. A reduced version of the grammar was implemented: human shorthand notation was not included.

- 2.2. The AST is then traversed applying transformation rules in order to generate the code equivalent of the constraint. Section 4.3 describes the basic ideas behind the transformation approach taken in this work while Chapter 6 discusses this in detail. Note that this requires that some information be retrieved from the SUS source code (e.g., attribute Java types). Section 8.1.2 discusses how the ocl2j tool does this with SUS Java class files.
3. The assertion code for all the constraints is placed into an aspect. Section 4.2 discusses the advantages of externalizing assertion code from the SUS source code while Chapter 7 discusses how aspect oriented programming technology is used in order to achieve this.
 4. The aspect code is woven with the SUS code. Appendix E shows how this is done with respect to the ocl2j tool implemented with this approach.
 5. The constraints (in the form of assertion code) are checked while the instrumented SUS executes (at runtime). Section 4.4 discusses exactly when the constraints are checked in the code. Upon a violated assertion failure the user is notified by a runtime exception being thrown or an error message being printed to the standard error output stream. This is discussed in Chapter 8.

It is worth noting that the only work reused from the toolkit is (SableCC) OCL grammar.

4.2 Externalization of Contracts

Aspect code resides in its own files (outside of the SUS's source code), leading to the following advantages:

- The user can work on the code without having to regenerate the contracts before each compile. This is true as long as the class diagram and contracts do not change as then the contract code will not change either. This saves the user large amounts of time as the generation and instrumentation of contracts with the toolkit has shown to take significant amounts of time.
- Having the program code and the contract code in separate files allows for parallel development (as long as class interfaces remain unchanged). As one person is working on the program source code the other person can be creating the contract code. The two are later woven together.
- When dealing with version management software and using the toolkit, the user must store two versions of the program code: a version with the contracts and a version without the contracts. With ocl2j there is only one version of the code in the version management software's database, thus avoiding duplication. This is a big advantage as duplication often leads to inconsistencies and cleaning of the code takes a significant amount of time.

- If the user wants to run the program without the contracts it's as easy as recompiling the code without including the compiled aspects.
- The weaving of the contract code and the program code is done at the byte code level, for this reason the source code is not necessary for the insertion of the contract code. This is very important if the SUS uses classes for which the source code is not available.

An introduction to aspect oriented programming technology was given in section 2.4 while Chapter 7 describes in detail how this technology is within the context of this work.

4.3 OCL to Code Transformations

As discussed in section 3.4, the toolkit treats all types equally: no matter if a Java variable is an integer or a collection, both will be converted to an OCL equivalent type. This is not only unnecessary but also very inefficient. The only advantage of this approach is that it provides for straightforward code generation logic. On the other hand ocl2j analyzes the code statically so that the generated code is as efficient as possible. This section gives an overview of the approach (in comparison to the approach in the toolkit) and lists its benefits, while detailed information regarding this topic is found in Chapter 6.

Consider the following example:

A class has a method `add(i1: int, i2: int): Integer` where `i1`, `i2`, and the result are both of Java primitive type `int`. This method has the following postcondition: `result = i1 + i2`. To check this constraint the toolkit would first convert `result`, `i1`, and `i2` to object of type `java.lang.Integer`, then it would check the constraint with code like this: `objectResult.equals(objectI1.add(objectI2))`. Thus, the checking of this constraint costs the creation of three objects plus the invocation of two operations.

`ocl2j` would detect that `result`, `i1`, and `i2` are all of type `int` and it would generate the following code to check the constraint: `(result == (i1 + i2))`. Such a system creates no unnecessary objects nor does it execute any unnecessary methods.

Thus, the benefits of this system are threefold:

- When possible, OCL operations are executed directly on the variables. This is sometimes not easy as intermediate collections may have to be generated. For example, the OCL collection operation `someCollection->including(someObject)` (that returns a collection with all the elements found in `someCollection` plus `someObject`) requires for a new collection to be created before further operations can be performed on that collection. In contrast to the collection operation `someCollection->size()` which maps directly to the `size()` method of the

`java.util.Collection` interface (which every collection in Java implements).

- Some OCL Integer and Collection operations cannot be directly converted to operations found in the standard Java libraries, for instance the `someCollection->count(someObject:OclAny):Integer` operation (that counts the number of times `someObject` occurs in `someObject`). The aspect contains inner classes with operations that provide this missing functionality. To continue the count example the inner class `OclCollection` has a `count` method that takes two arguments: the collection on which count must be performed and the object that needs to be counted. This has two consequences:

- No wrapping of the Java objects is ever necessary.
- No modification of the source code is necessary.

The one drawback of this approach is that static analysis of the SUS code must be performed. The disadvantage is that the process of generating the assertion code may take longer, but this is not a significant disadvantage as the static analysis is only done once at the preparation stage. Section 8.1.2 describes the means employed by `ocl2j` in order to perform static analysis of Java class files.

4.4 Checking of Contracts

The insertion point for precondition checking code is right before a method starts executing. The insertion point for postcondition checking code is right after a method finishes executing. In [12] it is specified that “the invariant must be true upon completion of the constructor and every public method but not necessarily during the execution of methods”, and this is the definition we follow.

4.5 Inheritance of Constraints

Since a subclass inherits features from its superclass it's natural to assume that the same should happen with constraints. The Liskov Substitution Principle (LSP) [35] states that “if for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behaviour of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T ”. If this principle is followed then the following is true in terms of constraint inheritance:

- A subclass invariant implies the superclass invariant.
- In case of methods:
 - The precondition is weakened.

- The postcondition is strengthened.

Having said this, it is a reality that the LSP is not always followed. Thus, ideally the user should be allowed to specify the inheritance hierarchies where the LSP is and is not followed so that constraint inheritance is only enforced where it makes sense to do so.

In this work constraint inheritance is only supported in the case of invariants (due to time restrictions) and the user has the option to instrument the SUS with constraint inheritance support for all inheritance hierarchies or none. Appendix E shows how to do this with the ocl2j tool. OCL does not specify any explicit rules with respect to the inheritance of constraints.

4.6 Assumptions

This section lists the assumptions made by the ocl2j tool.

4.6.1 OCL Syntax Reduction

To facilitate the processing of OCL expressions a subset of the OCL syntax is used. The OCL syntax not taken into account is human shorthand syntax (like the `let` expressions). An OCL expression using the full OCL syntax can be easily transformed to only use the syntax allowed by ocl2j with a normalizer (like the one found in the toolkit, see section

3.3).

The restrictions are as follows:

- No use of shorthand in the OCL expression of a collection operation: the iterator and its type must be specified and used through the expression (i.e. the iterator cannot be used implicitly).

Incorrect: `collectionOfPeople->collect(age > 18)`

Correct: `collectionOfPeople->collect(p:Person|p.age > 18)`

- No use of shorthand in place of the `collect` collection operation.

Incorrect: `collectionOfPeople.age`

Correct: `collectionOfPeople->collect(p:Person|p.age)`

- The navigation across multiple collections without the use of `collection->collect(expr:OclExpression):Collection(expr.evaluationType)` is not allowed.

Incorrect: `someCollection.someOtherCollection->size()`

Correct: `someCollection->collect(e:ElementType|`

`e.someOtherCollection)->size()`

- Only `self` is used to refer to the context class.

Incorrect: `context c : Company inv: c.numberOfEmployees > 50`

Correct: `context Company inv: self.numberOfEmployees > 50`

- All variable types must be specified.

Incorrect: `collectionOfPeople->collect(p|p.age)`

Correct: `collectionOfPeople->collect(p:Person|p.age)`

- No use of the infix notation.

Incorrect: `a.+(b)`

Correct: `a + b`

- The qualified version is used in the navigation to an association. This is explained in detail in appendix A.3.1.
- The context must always be specified.

- `let` statements are not supported.

From this list the only restriction that can potentially make the generated code less efficient is the lack of support for the `let` statement. Furthermore, this will only happen in the case where `let` is used to define an expression that takes a long time to compute and the definition is then used multiple times in the constraints.

4.6.2 `forAll` Collection Operation

As stated in [6], “The `forAll` operation has an extended variant in which more than one iterator is used. Both iterators will iterate over the complete collection.” In this paper this is interpreted as saying that the OCL collection operation `forAll` uses a maximum of two iterators (the transformations only take into account a `forAll` with one or two iterators).

4.6.3 `collection->sum() : T`

This operation is only applicable to collections where the element type implements the `java.lang.Number` interface. This requirement is equivalent to the requirement in [6] specifying that “elements must be of a type supporting the ‘+’ operation.” In the standard Java 1.4 library the following types fulfill this requirement: `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`.

4.6.4 collection->sortedBy(expr : OclExpression) : Sequence(T)

The type of the `expr` must be a primitive type, or an object implementing the `java.util.Comparable` interface. This is equivalent to the requirement in [6] specifying that “the type of the `expr` expression must have the ‘<’ operation defined.”

4.6.5 User Defined Collections Used in OCL Constraints

User defined collections must implement the `java.util.Collection` directly or indirectly. Additionally, sequences and bags must implement the `java.util.List` interface while sets must implement the `java.util.Set` interface.

4.6.6 Protected & Default Access Methods

In Java protected and default access methods can be publicly accessed from within the package, thus it may be argued that from the invariant checking perspective protected methods should be treated in the same manner as public methods. This work uses the UML definition of when invariants should be checked (see section 4.4), thus invariants are not be checked before and after protected and default access methods.

4.7 Limitations

This section lists oclj2's limitations followed by the two subsections specifying properties of OCL that could not be implemented.

oclj2's limitations:

- The subset of the OCL syntax must be used as discussed in section 4.6.
- Only one stereotype (inv/pre/post) is allowed per context.
- The following point is only applicable when specifying a constraint on an operation (any pre- or postcondition). In the case of OCL collection operations where the parameter is an `OclExpression` the iterator name can't be the same as the method's parameter names.
- When specifying the parameter types of a constraint operation, or the return type of that operation, the type must be the Java type used in the implementation. For example, let's say you have the following constraint:

```
context Person::getAge():Integer post: result = self.age
```

In this case the return type of the method `getAge()` (`Integer`) would be specified with the Java type used in the SUS code, for example the primitive type `int` or the object type `java.lang.Integer`. This reason for this limitation is a pragmatic one. In the

current mainstream UML modeling tools (like Borland Together) the types used in the model correspond to the type in the implementation language. Furthermore, the context declaration is automatically generated by the tool. Thus, it is simpler for the tool to just specify the implementation language type and it's simpler for the ocl2j tool to work with constraints that have the implementation language type specified. This is not a serious limitation and can easily be eliminated once the need arises (by the arrival of tools that support development at a higher level of abstraction).

None of these limitations are highly constraining in practice as a normalizer (like the one found in the toolkit, see section 3.3) can act as the front-end for ocl2j and normalize the constraint to the above listed constraints for the user automatically.

Additionally, as discussed in section 4.5, constraint inheritance is only supported in the case of invariants.

4.7.1 Enumerations

Enumerations are not supported as there is no Java construct for enumerations making them implementation specific. Consider the enumeration example in [6], where the gender of a person is an enumeration: `enum {male, female}`. When it comes to the implementation there is no standard way to implement this. If done properly this is implemented with the enumeration idiom presented in Item 21 of [14] making the

enumeration look like this:

```
public class Sex {
    private final String sex;

    private Sex(String sex) {
        this.sex = sex;
    }

    public String toString() {
        return sex;
    }

    public static final Sex MALE = new Sex("Male");
    public static final Sex FEMALE = new Sex("Female");
}
```

While it can also be implemented in this manner:

```
public class Sex {
    public static final int MALE = 0;
    public static final int FEMALE = 1;
}
```

This kind of freedom of implementation makes support for the OCL enum keyword non-trivial (how does ocl2j know where to find the implementation for the enumeration?), while adding little to enriching the language (in the constraint the enumeration can be simple written in the `Sex.MALE` / `Sex.FEMALE` manner). This situation will be alleviated with the arrival of Java 1.5 [23] as that release has primitive support for enumerations. It is worth noting that if ocl2j was to be integrated with a CASE tool like IBM Rational Rose [24] then enumerations could be easily supported as the CASE tool would be the one generating the code for the enumeration, thus knowing the idiom used to implement the enumeration.

4.7.2 OclState and oclInState

The OCL documentation specifies the operation `oclInState(s:OclState):Boolean` as one predefined on all objects. Similarly to the case of enumerations there is no one way to implement this in Java. Additionally, this functionality is very tersely specified. For this reason the only way to provide support for this functionality is with the integration of `ocl2j` with a CASE tool that knows how the state properties of objects are implemented and used (for example using the “state pattern” as specified in [34]).

4.8 Scalability

The scalability of the `ocl2j` is a very intricate issue as experiments with the tool have shown (this will be discussed in detail in chapters 9 and 10). It was found that, in certain circumstances, the approach of transforming OCL collection operations directly to the Java collection operations contributed most to performance degradations in the instrumented SUS. How much of an overhead is generated by the constraint checking and enforcement code depends on several factors: the size of the collection, the type of the collection, the arrangement of the elements in the collection, the complexity of the elements in the collection, and the type of OCL operations used on the collection.

Consider the following constraint:

```
arrayListCollection->includes(someObject)
```

where `arrayListCollection`, of OCL type `Sequence`, is a very large collection which has only one occurrence of `someObject`. If `someObject` occurs at the beginning of the collection then a direct transformation to the `java.util.Collection::contains(o:Object):Boolean` method will outperform any optimization technique. This is due to the fact that these optimization techniques introduce overhead (for example, logic must be added to observe the collection for changes) whereas the `java.util.Collection::contains(o:Object):Boolean` would return “true” after looking at the first few elements in the collection. On the other hand, if `someObject` occurs at the end of the list then optimization schemes can easily outperform the direct transformation approach. This simple example shows the complexities that must be faced when trying to optimize OCL constraint operations on collections. Chapter 10 provides a detailed discussion on the proposed solutions (backed by preliminary results) with respect to this issue.

Apart from this issue no other threats to scalability were discovered.

5 THE OCL PARSER⁸

The first step towards the processing of an OCL expression is a transformation of the expression into a syntax tree by a parser. The Java code generator then takes the syntax tree generated by the parser as its input and traverses it generating the Java code. This chapter discusses the compiler compiler used for the generation of the parser, the generation of the abstract syntax tree by this parser, and the tree walkers used to traverse the abstract syntax tree.

5.1 SableCC – An Object-Oriented Parser Generator

For the development of parsers, a vast variety of tools exist, called compiler compilers. Some of the best-known tools that generate parsers in the Java language probably are JavaCC [28] and ANTLR [29]. Such tools generate the source code for a parser out of a grammar description. For the OCL Parser, the SableCC compiler compiler [4] was chosen. In contrast to other compiler compilers, SableCC does not mix the grammar specification with action code (in this case the action code is the java code generator), but

⁸ Some parts of this section paraphrase material found in [2].

rather it produces a compiler framework with a parser and tree walkers. The generated parser produces an abstract syntax tree. The classes whose objects constitute the abstract syntax tree are themselves generated automatically from the grammar specification. The tree walkers traverse the abstract syntax tree and can be adapted to handle different tasks through subclassing. A description of tree walkers, which are used to implement AST traversals, is given in section 5.3.

It was decided to use SableCC in this work since an OCL 1.3⁹ grammar was already written for it. The original version of this SableCC grammar was written by the toolkit authors and it was modified in this work. The grammar can be found in appendix C.1. The modifications and problems with respect to the grammar are discussed in appendices C.2 and C.3.

In the SableCC grammar, production rules are specified in EBNF-like syntax¹⁰. Production rules define phrases and are composed of tokens, and subphrases (other, already defined, phrases). Token symbols (or just tokens) are atomic symbols used in the language, for example in Java these are 'if' and ';'.

⁹ ocl2j accounted for a problem with the grammar where packages cannot be specified in certain situations (for example: when specifying the context of an operation or when specifying the type of a parameter variable).

¹⁰ Please consult [4] for reasons as to why the exact EBNF syntax was not used.

Take the following (OCL) production rule:

```
implies_expression = logical_expression implies_expression_tail* ;
```

In this case `implies_expression` is the phrase, `logical_expression` and `implies_expression_tail` are the subphrases, and '=', ' ', ';' are the tokens.

The '*' operator trailing the `implies_expression_tail` subphrase is part of set of regular expression operators which augment the BNF syntax in order to define the EBNF syntax. Table 2 summarizes these operators and their functionalities.

Operator	Definition
*	The subphrase can occur zero or more times.
?	The subphrase occurs either once or not at all.
+	The subphrase can occur one or more times.

Table 2: Definitions of the regular expression operators BNF is augmented with in EBNF

5.2 Syntax Trees

The OCL parser generates an abstract syntax tree. The nodes of this tree are instances of classes that are automatically generated by SableCC from the OCL grammar. Every production rule `production_name` of the grammar is transformed into an abstract class `PProductionName`, every alternative `alternative` of this production rule is transformed into a concrete subclass `AAlternativeProductionName` of the production class. A class `TTokenName` is generated for a token `token_name`. If a part of the right-hand side of the production rule may occur multiple times (denoted by an asterisk, '*'), this is mapped to an attribute of the corresponding alternative class that is of the type `java.util.LinkedList`.

In this thesis, classes generated for grammar productions will be called production classes, while classes generated for grammar alternatives are called alternative classes.

The following examples illustrate these generation patterns.

Example 1

The production rule “`postfix_expression = primary_expression postfix_expression_tail*` ;” is represented by the class diagram in Figure 5-1 (attributes and operations have been omitted). This production is considered to have only one alternative, hence the presence of the `APostfixExpression` class in the class diagram.

Figure 5-1: Classes generated for the production `postfix_expression`

Example 2

The production rule “add_operator = {plus} '+' | {minus} '-';” is represented by the class diagram in Figure 5-2 (attributes and operations have been omitted). The names within the braces (‘{’, ‘}’) are the names of the alternative. These names are used in the generation of the node classes, as can be seen in the class diagram.

Figure 5-2: Classes generated for the production add_operator

5.3 Tree-Walkers

One of the basic functionalities required to work on an AST is visiting its nodes. A tree-walker is a class that will visit all the nodes of an AST in a predefined order. This section describes SableCC's tree-walkers in detail as they are used extensively in this work.

5.3.1 The Visitor Pattern

The tree-walkers employ an extended version of the Visitor design pattern as defined in [34]. Briefly, the visitor pattern can be used as an object-oriented solution to the problem of choosing actions depending on the class of an object. A straightforward (Java) solution to this problem would use the `instanceof` operator:

```
Shape shape=...;
if (shape instanceof Circle) {
    // ... actions for Circle
} else if (shape instanceof Rectangle) {
    // ... actions for Rectangle

    // This continues for all the shapes.
}
```

This approach is not runtime efficient and hard to extend with new types.

In Figure 5-3, an extension of the visitor pattern is applied and an interface `ShapeSwitch` is used to solve this problem (attributes have been omitted from the class diagram). The example code from above could now be rewritten as follows; using an anonymous inner class (the anonymous class or any class implementing `Switch` is called a visitor):

```
Shape shape = ...;
shape.apply(
    new ShapeSwitch() {
        public void caseCircle(Circle c) {
            // ... actions for Circle
        }
        public void caseRectangle(Rectangle r) {
            // ... actions for Rectangle
        }
        // etc ...
    } // end inner class
);
```

Figure 5-3: An example of the visitor pattern

Note that the `apply` method defined in the abstract class `Shape` calls the appropriate `case` method in the `ShapeSwitch` implementation.

The visitor pattern suffers from some problems, it is hard to add new element types to the visited structure, and visiting across class hierarchies is impossible. The SableCC authors made some modifications to the visitor design pattern to overcome these limitations and render the design usable in the context of evolving structures. This is done by:

- Making the switch interface very generic (it does not have any operations) and is the ancestor of all the switch interfaces in the framework.
- A Switchable interface is introduced that every switchable class implements (like Circle and Rectangle).

The introduction of the Switch and Switchable interfaces allows for the addition of a new Oval shape without modifying any of the existing classes:

```
interface ExtendedShapeSwitch extends Switch {
    void caseOval(Oval obj);
}

class Oval extends Shape {
    // class code

    void apply(Switch sw) {
        (ExtendedShapeSwitch) sw.caseOval(this);
    }
}
```

To add a new shape (or a collection of new shapes), a new interface is defined that extends switch and includes a case method for each new shape (caseOval in this case). Then, each new shape class implements the Switchable interface. To account for this new shape a new interface is defined:

```
interface AllShapesSwitch extends ShapeSwitch, ExtendedShapeSwitch {}
```

This is later used in the following manner:

```
void Selected(Shape obj) {
    obj.apply(new AllShapesSwitch() {
        void caseCircle(Circle obj) {
            System.out.println("a circle was selected");
        }
        void caseRectangle(Rectangle obj) {
            System.out.println("a rectangle was selected");
        }
        void caseOval(Oval obj) {
            System.out.println("an oval was selected");
        }
    });
}
```

Note that the only changes the introduction of a new shape has inferred are the addition of the `AllShapesSwitch` interface and the expansion of the inner class logic. `ShapesSwitch` did not have to be modified.

5.3.2 Visitors and Tree-Walkers in SableCC

The SableCC-generated framework defines an `Analysis` interface that extends `Switch`. This interface contains all the `caseXxx` methods for token classes (`TXxx` and `Token`) and alternative classes (`AXxx` and `Start`). Naturally, class `Node`, the ancestor of all AST classes, implements `Switchable`. These interfaces are shown in Figure 5-4, together with the class `AnalysisAdapter`. This utility class implements `Analysis` to call a method `defaultCase` in all methods, which is an empty method. Subclasses of `AnalysisAdapter` can overwrite a subset of the methods of `Analysis` and give a standard alternative in `defaultCase`.

Figure 5-4: The visitor pattern in the SableCC framework

SableCC generates the `DepthFirstAdapter` tree-walker class which traverses the abstract syntax tree in a recursive depth-first (top-down) traversal. All the `caseXxx` (`Xxx` stands for the alternative) methods for the alternatives in `AnalysisAdapter` are overwritten in order to call a new method `inXxx`, and then apply the traversal to all subnodes (left-to-right), and then call a method `outXxx`. The traversal to all subnodes is accomplished using the `apply` method. Once the `apply` method is invoked on a node the reference to the tree walker is also passed to the node. The node then invokes the method appropriate to that node on the tree-walker (achieving recursion).

This is shown in Figure 5-4 for the `APostfixExpression` alternative. The default implementations of `inXxx` and `outXxx` call `defaultIn` and `defaultOut`, respectively, which are empty. The `caseXxx` methods for token (e.g. `caseTPlus`) of `AnalysisAdapter` are not overwritten and therefore still call `defaultCase`.

The logic for `APostfixExpression`'s case method is:

```
public void caseAPostfixExpression(APostfixExpression node) {
    inAPostfixExpression(node);
    if (node.getPrimaryExpression() != null) {
        node.getPrimaryExpression().apply(this);
    }
    Object temp[] = node.getPostfixExpressionTail().toArray();
    for (int i = 0; i < temp.length; i++) {
        ((PPostfixExpressionTail) temp[i]).apply(this);
    }
    outAPostfixExpression(node);
}
```


This case method invokes the apply method on objects of type PPostfixExpressionTail. PPostfixExpression is an abstract class while APostfixExpression extends this class (refer to Figure 5-1) and provides logic to the apply method. The apply method looks like this:

```
public void apply(Switch sw)
{
    ((Analysis) sw).caseAPostfixExpression(this);
}
```

This class walks the tree in a recursive top-down traversal, calling the inXxx and outXxx methods on entering and leaving nodes. But more interestingly, since the case methods already provide the tree-walking logic, a subclass of DepthFirstAdapter only has to overwrite the inXxx and outXxx methods in order to add action code to the tree-walker.

The Java code generator developed in this work extends the DepthFirstAdapter class and overwrites the required inXxx and outXxx methods in order to add the action code.

For example, the following code processes an OCL string literal in a constraint:

```
public void inAStringLiteral(AStringLiteral sl) {
    String lit = sl.toString().trim();
    // Convert the OCL string literal to a Java string literal
    lit = lit.replaceAll("'", "\\'");
    // Process the now transformed Java string literal
    appendCode(lit);
}
```

This code converts an OCL string literals like 'some string' to Java string literals, in this case the result would be: "some string". A more elaborate example is given in section 6.1, after transformation rules are explained.

6 THE OCL TO JAVA TRANSFORMATIONS

This chapter begins by giving an overview of the transformation rules in section 6.1. The remaining sections discuss important issues with regards to the transformation rules. More specifically, section 6.2 discusses the effects that Java's dual type system has on the transformations. Section 6.3 discusses the topic of object in equability from the OCL and Java points of view. Section 6.4 discusses discrepancies between OCL and Java collections, and finally section 6.5 discusses some subtleties that stem from the usage of the OCL `@pre` keyword with respect to Java programs. The tool utilizes 48 transformation rules.

6.1 Transformation Rules and Usage

The Java code generator transforms an OCL constraint represented as an abstract syntax tree into a Java code assertion. To achieve this, the code generator uses transformation rules. These rules, along with their usage are described in this section.

Each transformation rule is described in the format presented in Table 3.

1	Production rule using EBNF-like syntax.
2	Condition for the invocation of the rule using some OCL syntax.
3	Generated preparation Java code.
4	Generated assertion java code piece.
5	Comments.

Table 3: Transformation Rule Table

1. A production rule defined within an EBNF-like syntax, the definition of production rules and the syntax are discussed in Chapter 5.
2. The condition that must be fulfilled for this transformation to take place using EBNF and some OCL syntax on a subphrase in the production rule. Table 4 defines the OCL syntax used and its meaning in relation to the subphrase it's used with:

OCL Syntax	Interpretation
path_specified_by_the_ subphrase->size()	The cardinality of the path specified by a subphrase.

Table 4: OCL Syntax Used in Conjunction with Subphrases

For example, for the transformation defined in Table 5 to take place we must have an OCL constraint in the form where some “logical expression” implies at least one other “logical expression”, hence for the transformation to take place `implies_expression_tail` must be present (ensured by the `notEmpty()` operation).

1	<code>expression = logical_expression(1) implies_expression_tail* ; implies_expression_tail = "implies" logical_expression(2) ;</code>
2	<code>expression = logical_expression(1) implies_expression_tail+ ;</code>
3	<code>boolean bImplies = logicalExpr1Code; boolean b2Implies = logicalExpr2Code;</code>
4	<code>(!bImplies (bImplies && b2Implies))</code>
5	The logicalExpression1Code & logicalExpression2Code is generated by another rule during the recursive top-down traversal of the AST.

Table 5: The expression transformation

3. Sometimes some preparation code is needed before the assertion code can execute. A simple example of this is shown in Table 8. Consider the `java.awt.Point` class where we may want to put the constraint “`self.x@pre = self.x` and `self.y@pre = self.y`” on every query method (in case we want to be extra careful that we don’t introduce any errors). In this case the values of `x` and `y` must be saved before the method executes (as the assertion code will need these values).
4. The piece of code that this transformation rule generated that contributes directly to the assertion; this is built incrementally as the AST is traversed. Once the assertion is constructed it is in the form where it returns a Boolean upon evaluation. Later, during the execution of the SUS, the assertion is tested using a Java if statement.
5. Any additional explanation details and information regarding the transformation is placed in the comments section.

The usage of the transformation rules will now be exemplified on the following rules while a complete list of the rules is located in Appendix D.

1	logical_expression = equivalence_expression(1) logical_expression_tail* ; logical_expression_tail = logical_operator equivalence_expression(2) ;
2	logical_expression = equivalence_expression(1) logical_expression_tail+ ;
3	
4	(equivalenceExpr1Code logical_operator equivalenceExpr2Code)
5	The following transformation occur on the logical_operator: OCL -> Java and -> && or -> xor -> ^

Table 6: The logical_expression transformations

1	primary_expression = path_name time_expression? qualifiers? feature_call_parameters? ;
2	primary_expression = path_name;
3	
4	pathName
5	Occurrences of the OCL path resolution token ('::') and OCL collection feature call token ('->') are converted to the Java token: '.'.

Table 7: The primary_expression transformation¹¹

¹¹ This is the simplest condition for this transformation rule as only the path_name is specified.

1	<code>primary_expression = path_name time_expression? qualifiers? feature_call_parameters? ;</code>
2	<code>primary_expression = path_name time_expression+; and path_name->size() = 1</code>
3	<code>pathNameType pre = pathName;</code>
4	<code>pre</code>
5	In this case path_name points to a field. The variable pre is unique. This preparation code is executed before the constrained method executes.

Table 8: The primary_expression with time_expression transformation

The usage of these rules will now be exemplified by means of the class `ClassWithBooleans` which contains three attributes of type `boolean`; `b1`, `b2`, and `b3`. Please note that all transformations occur in the context of Table 5 as that rule defines the top-level production rule (`expression`).

Example 1: The simplest example.

```
context ClassWithBooleans inv: self.b1
```

Only the transformation rule defined in Table 7 (row 4) is used to produce the following Java assertion code “`(self.b1)`” that is evaluated with an if-statement. The rule defined in Table 7 is also used in all of the following examples but is not mentioned explicitly. Please note that `self` refers to the object on which the constraint is being validated upon. How this is achieved will be seen in Chapter 7 (Section 7.1). This transformation step will be elided from the following examples.

Example 2: `expression = logical_expression implies_expression_tail;`

context `ClassWithBooleans` inv: `self.b1 or self.b2 implies self.b3`

- The predicate begins with a logical expression “`self.b1 or self.b2`” thus the Table 6 (row 4) rule is employed to generate the following code (this happens in the context of rule Table 5, row 1): `(self.b1 || self.b2)`

- After transformation of `logical_expression(1)` as defined in Table 5 (row 1) is completed the `implies_expression_tail` is transformed. This transformation generated the following preparation code (that must be executed before the predicate can be checked – Table 5, row 3):

```
boolean bImplies012 = (self.b1 || self.b2);  
boolean b2Implies0 = (self.b3);
```

- Variables from the preparation code are then used in the assertion code (Table 5, row 4):

```
(!bImplies0 || (bImplies0 && b2Implies0))
```

¹² Every pair of `bImplies` and `b2Implies` is followed by an incrementally larger integer. The integer is incremented every time a new pair is generated.

Example 3: `expression = logical_expression implies_expression_tail+;`

```
context ClassWithBooleans inv: self.b1 implies self.b2 implies self.b3
```

This constraint is quite similar to the last one; the only difference is that there are multiple occurrences of `implies_expression_tail`, thus each `implies_expression_tail` is sequentially analysed.

1. First, code for `self.b1 implies self.b2` is generated.

Preparation code (Table 5, row 3):

```
boolean bImplies0 = (self.b1);  
boolean b2Implies0 = (self.b2);
```

Assertion code (Table 5, row 4):

```
(!bImplies0 || (bImplies0 && b2Implies0))
```

2. Next, the second `implies_expression_tail` is analysed.

Preparation code (Table 5, row 3):

```
boolean bImplies1 = (!bImplies0 || (bImplies0 && b2Implies0));  
boolean b2Implies1 = (self.b3);
```

Assertion code (Table 5, row 4):

```
(!bImplies1 || (bImplies1 && b2Implies1))
```


Example 4: The use of the recursive depth-first descent.

context ClassWithBooleans inv:

```
self.b1 implies (self.b2 implies self.b3)
                |-----expression 2-----|
|-----expression 1-----|
```

In this case the brackets (used to specify precedence) around the second implies expression make the first expression a recursive one. It is recursive in that the result from the second implies expression needed in order to evaluate the first. Note that in both cases `implies_expression_tail->size() = 1`. The `inAImpliesExpression` method is executed at each implies expression analysis entry and `outAImpliesExpression` at each exit (see Section 5.3.2 for details). It will now be shown how these methods are used in order to deal with recursion.

1. The code for `self.b1` is generated.
2. The `implies_expression_tail` is detected invoking the `inAImpliesExpression` method.
 - 2.1. The generated code (`self.b1`) is placed on a stack.
 - 2.2. The code for the second implies expression is generated:

Preparation code:

```
boolean bImplies0 = (self.b2);  
boolean b2Implies0 = (self.b3);
```

Assertion code:

```
(!bImplies0 || (bImplies0 && b2Implies0))
```

3. After processing the `implies_expression_tail` the `outAImpliesExpression` method is invoked.

3.1. The code from the stack is put into `bImplies1`.

3.2. The most recently generated (assertion) code is put into `b2Implies1`.

3.3. The new assertion code is generated.

After these steps the following preparatory code generated:

```
boolean bImplies0 = (self.b2);  
boolean b2Implies0 = (self.b3);  
boolean bImplies1 = (self.b1);  
boolean b2Implies1 = (!bImplies0 || (bImplies0 && b2Implies0));
```

And the following assertion code is generated:

```
(!bImplies1 || (bImplies1 && b2Implies1))
```

6.2 Resolutions of Java Types

In addition to object types (like `Integer` and `Long`) Java also supports primitive types (like `int` and `float`). OCL on the other hand has no notion of primitive types as everything is considered an object. This has a major impact on the process of transformation of OCL constraints into Java code. For example, consider the following OCL constraint:

```
someCollection->includes(5)
```

In OCL “5” is an object and OCL Integer operations can be invoked on this “object”. To achieve the intended effect in Java the primitive integer 5 must be wrapped by the `int` object wrapper (`new Integer(5)`) as Java collections only take objects as elements.

A trivial solution to this problem would be to just convert everything into an object, but as discussed in section 4.3, this makes for an inefficient solution. Thus, when transforming OCL constraints into Java code the type of the parameters used in the OCL expression (object or basic) must be kept track of. Since Java uses both reference (i.e. `java.lang.Integer`) and primitive (`int`) types to represent OCL basic types¹³, objects must be converted to their primitive types and vice versa when necessary. During the

¹³ Except for the OCL Basic type `String`, in Java `String` is only represented by a reference type.

transformation of OCL constraints into Java code the following two algorithms (later exemplified) are used:

- In the case of parameters used in logical, addition, multiplication, and unary operations: the parameters are evaluated in their primitive form. If one or both of the parameters is in reference form then the primitive equivalent is extracted from it through a method invocation. For example, in the case of a parameter being an instance of the `java.lang.Integer` class (reference form), the primitive value of the integer would be retrieved using the `intValue()` method.
- In the case of class method invocations, a parameter of basic type is automatically converted into the form required by the method. For example, the `java.util.Collection::contains(o:Object):Boolean` operation expects parameters to be in reference form. Thus, in the case of the OCL constraint `integerCollection->contains(10)`, where `integerCollection` is of type `java.util.Collection`, "10" is converted to its reference form using the integer object wrapper: `new Integer(10)`.

Consider the constraint `"self.someCollection->including(10)->size() = self.collectionSize"` where `someCollection` is of type `java.util.HashSet` and `self.collectionSize` is of type `java.lang.Integer`. The following steps are taken to generate the assertion code for this constraint:

1. The left-hand-side and the right-hand-side are of Integer type, as already discussed in Java Integers can be expressed in primitive form, thus the two sides are equated in a primitive context using the Java “==” operator.

1.1. A new collection is created and all the elements from `self.someCollection` are added to this new collection, say `someNewCollection`.

1.2. The operation `including` is being used which expects a parameter in reference form, thus this Integer parameter must be converted from primitive form to reference form.

1.2.1. The Integer 10 is wrapped and the addition of this element to the collection looks like this: `someNewCollection.add(new Integer(10))`.

1.3. The collection operation `size()` returns an Integer of primitive type, thus it does not need to be altered.

1.4. The right-hand-side of the equals operations `self.collectionSize` is of type `java.lang.Integer`, thus `self.collectionSize` is transformed to its primitive form: `self.collectionSize.intValue()`.

The preparation code for this constraint is:

```
java.util.HashSet someNewCollection = new java.util.HashSet();
someNewCollection.addAll(someCollection);
someNewCollection.add(new Integer(10));
```

The assertion code looks is:

```
(someNewCollection.size() == self.collectionSize.intValue())
```

The following table lists the Java types equivalent to the OCL Integer, Real and Boolean types.

OCL	Java
Integer	Objects: Integer, Long, Short, Byte Primitives: int, long, short, byte
Real	Objects: Double, Float Primitives: double, float
Boolean	Objects: Boolean Primitives: boolean

Table 9: Integer, Real & Boolean types in OCL and Java

Furthermore, when in the presence of query operations in OCL constraints, further complications may arise. When using Integer and Real types as actual parameters in operations in the UML model the source code is used in the retrieval of the Java parameter and return type of the method. For example, consider a query method `SomeClass::add(i1:Integer, i2:Integer):Integer` and the following OCL expression using this query method: `add(1, 2) = 3`. When trying to determine the return type of the method we would first try to find a method with the name “add” and with parameter types “int, int” based on the way add is used in the constraint. If this straightforward search fails a more advanced search is performed for a method with the same name, the same number of parameters, but other variations of the (Java) Integer and Real types. This exemplifies why our approach requires some limited static code analysis.

6.3 Object Equality

In OCL two objects (`object` and `object2`) are equal if, as defined in [6], “`object` is the same object as `object2`.” In Java, equality between objects is determined via the `equals(o:Object):boolean` operation. In the case of that the parameters being equated are in reference form and they have a primitive form, the Java “`==`” operator is used to determine equivalence.

For example, two points `p1` and `p2`, both of type `java.awt.Point`, are tested for equality in the following manner: `p1.equals(p2)`. On the other hand, the integers `i1` and `i2`, both of type `java.lang.Integer`, are tested for equality in the following manner: `i1.intValue() == i2.intValue()`.

6.4 OCL Collections to Java Collections

In OCL there are three collection types: `Set`, `Bag`, and `Sequence`. These collections types do not have a one-to-one mapping to the Java collection types. Java has two main collection types: `java.util.Set` and `java.util.List`. The similarities and differences between OCL and Java collections will now be discussed.

6.4.1 OCL Bag and Sequence in Java

In addition to Set, OCL also has the Bag and Sequence collection types. They are defined in [6] as follows:

- A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times. There is no ordering defined on the elements in a bag.
- A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once.

The OCL Sequence maps well onto the Java List, whereas OCL Bag has no equivalent counterpart in Java. This thesis assumes that OCL Bag collections are implemented with a Java List. It is interesting to note that a Bag collection could be partially supported by adding the implementation for an `OclBag` class to the aspect.

6.4.2 Additional Collections Operations

Most OCL Collection operations map trivially onto the Java collection operations. In the simplest case the Java collection operation has even the same name as in the case of `size()`.

For cases where the transformation is not this trivial the class `OclCollection` (located in

the aspect, see section 8.2) has methods that perform these operations. For example, in the Java Collections framework there is no count method. Thus, the following code performs this functionality:

```

public static int count(Collection c, Object o) {
    if (c instanceof Set) {
        return countSet(c, o);
    } else {
        return countOther(c, o);
    }
}

public static int countSet(Collection s, Object o) {
    return s.contains(o) ? 1 : 0;
}

public static int countOther(Collection c, Object o) {
    int num = 0;

    for (Iterator i = c.iterator(); i.hasNext();) {
        Object e = i.next();
        if (e.equals(o)) {
            num++;
        }
    }

    return num;
}

```

The most complex transformation takes place when the OCL collection operation parameter is of type `OclExpression`. This is the case with the following operations: `exists`, `forAll`, `isUnique`, `sortedBy`, `select`, `reject`, `collect`, and `iterate`.

Thus far we have presented two manners in which we transform OCL collections operation to:

1. A direct mapping between the OCL collection operation and a Java collection operation (as is the case with the `size()` operation).
2. The `OclCollection` class provides OCL collection operations in Java (as is the case with the `count()` operation).

Neither of these solutions helps in this case, the first solution cannot be applied since there are no equivalent methods in the Java Collections framework, and the second solution cannot be applied either as the parameter list cannot be predicted. Consider the example where a collection has the following constraint:

```
collectionOfPeople->forAll(p:Person | p.name <> '')
```

This example demonstrates that each OCL collection operation with an `OclExpression` as the parameter can lead to different generated code. In `ocl2j` this problem is solved by transforming each such operation to a unique method in the aspect where the method contains the `OclExpression` translated into Java code. In the case of the `forAll` operation the following template is used to generate the code for this method:

```
(1) private boolean forAll#(Collection ocl2jCollection) {  
    for (Iterator ocl2jIterator = ocl2jCollection.iterator();  
        ocl2jIterator.hasNext();  
        ) {  
(2)         IteType iteName = (IteType) ocl2jIterator.next();
```

```

(3)          if (!(OclExpr predicate in Java Code format)) {
                return false;
            }
        }
    return true;
}

```

1. Each `forAll` method is appended with a unique integer to differentiate the methods.
2. The casting and the referencing of the iterator.
3. The predicate found in the `OclExpression` is checked for validity. This code is generated using the already defined mappings and transformations. The implementation allows for recursion as that simply introduces the creation of other similar methods in the aspect file.

In this case the aspect would contain this method:

```

private boolean forAll0(Collection ocl2jCollection) {
    for (Iterator ocl2jIterator = ocl2jCollection.iterator();
        ocl2jIterator.hasNext();
        ) {
        Person p = (Person) ocl2jIterator.next();
        if (!((p.name) != (""))) {
            return false;
        }
    }
    return true;
}

```

The mapping for the remainder of the operations is located in Appendix D.

6.4.3 Default Collection Type Equivalents in Java

Certain OCL operations return a collection as the result. For example, consider the `OclType::operations():Set` operation, it returns the set of operations of any class in the model. Thus it is necessary to define the Java collection types that will be used in such cases. After consulting [16] (a very good guide to the Java collections framework) the following standard java library classes are used to implement the OCL collection types in the aspect code. The best-performing implementations were chosen with respect to how they will be used by the contracts. In the case of Set, it was possible to use either `HashSet` or `TreeSet`. `HashSet` was chosen as it is much faster (time vs. log time for most operations), the only drawback is that it offers no ordering guarantees – something we do not require. In the case of Sequence, it was possible to use `ArrayList`, `LinkedList`, and `Vector`. `ArrayList` was selected as `LinkedList` is generally quite slow, and `Vector` is slower than `ArrayList` because it provides a synchronized access – whether you need it or not.

OCL Type	Default Java Equivalent
Set	HashSet
Bag ¹⁴	ArrayList
Sequence	ArrayList

Table 10: OCL/Java Collection Type Equivalencies

¹⁴ As discussed in section 6.4.1, the standard Java 1.4 Collections framework does not have a type equivalent to the OCL Bag type.

6.5 Using Previous Property Values in OCL Postconditions

This section discusses the practical implementation of the OCL language construct `@pre`, which allows accessing the previous value of an object property in the postcondition for an operation. In most cases this construct is handled in a trivial manner by simply creating a new variable of the same type and assigning it the value of the variable appended with the `@pre` before the constrained method executes, but one case requires special attention.

Consider an example where a class (`someClass`) has an attribute of type `Person` (the attribute name is `person`). The postcondition on `someClass::someMethod()` needs to specify that `someMethod()` will not change the state of object `person`. This can be specified in the following manner:

```
context someClass::someMethod() post:
    self.person@pre.equals@pre(self.person)
```

The important thing to notice here is that the constraint uses the `equals` method executed on the state of the object before `someMethod()` executed. The only way that this can happen is by making a clone of `self.person` prior to the execution of `someMethod()` and then invoking the `equals` method on the clone after `someMethod()` executes. As support for cloneability is not default to objects in Java (see `java.lang.Cloneable` in [13]), an OCL syntax analyzer should warn the user, at the time when the constraint is

being specified, of the consequences and preconditions to the use of @pre in such a manner. Thus, for the @pre keyword to be used in such a manner the object type (in this case Point) must be cloneable.

For a guide to providing support for cloneability see Item 10 in [14]. Essentially, two types of cloning methods exist. First, a shallow copy is where the fields declared in a class (if any) will have values identical to those of the object being cloned. In the case of a class exhibiting one or more composite relationships the shallow copy is not sufficient and a deep copy must be used. During a deep copy composite object must also themselves be cloned.

7 OCL2J AND AOP

This chapter uses the concepts introduced in section 2.4 to explain how assertion code is instrumented into the SUS bytecode and enforced dynamically (at runtime) in section 7.1. Section 7.2 describes how AspectJ was used to implement the OCL operations `OclType::allInstances()` and `oclAny::oclIsNew()`. Next, performance overhead issues due to the usage of this technology are discussed in section 7.3.

7.1 Usage of AOP in This Work

Once the OCL contracts are converted into Java code, this generated code must be integrated with the SUS code (this process is called instrumentation). As discussed in the related work chapter, the toolkit chose to insert this code into the SUS source code. This is not a practical solution for reasons discussed in section 3.6.

Thanks to the functionalities of AspectJ the generated code no-longer has to be inserted into the SUS's source code, instead, the code is now integrated at the bytecode level. The aspect (`Ocl2jAspect.java`) which contains the assertion code (for all constraints) is added to the SUS's default package.

The AspectJ code template for enforcing preconditions is:

```
before(SomeClass self[, parameters of the method - if any]:  
    execution([method return type] SomeClass.someMethod(  
        [parameter types of the method - if any]))  
    && target(self)  
    && args(raise)  
    && if (self.getClass() == SomeClass.class) {  
  
    // Check the postcondition.  
    if (!(predicate code)) {  
        constraintFailed("the constraint in OCL form");  
    }  
}
```

The AspectJ template code for enforcing invariants is:

```
private void inv[number](SomeClass self) {  
    if (!(predicate code)) {  
        constraintFailed("the constraint in OCL form");  
    }  
}  
  
// Before the execution of every public method in SomeClass  
before(SomeClass self) : execution(public * SomeClass.*(..))  
    && target(self) {  
    inv[number](self);  
}  
  
// After the execution of every public method in SomeClass  
after(SomeClass self) : execution(public * SomeClass.*(..))  
    && target(self) {  
    inv[number](self);  
}  
  
// After the execution of any constructor in SomeClass  
after(SomeClass self) : execution(SomeClass.new(..))  
    && target(self) {  
    inv[number](self);  
}
```


The AspectJ code template for enforcing postconditions is:

```
// Around the execution of the someMethod in SomeClass
someMethod return type around(SomeClass self
[, parameters of the method - if any]) :
    execution([method return type] SomeClass.someMethod(
        [parameter types of the method - if any]))
        && target(self)
        && args(raise)
        && if (self.getClass() == SomeClass.class) {

    // Create any @pre variables needed

    // Let the execution of the method proceed.
    [result =] proceed(self [, parameters of the method - if any]);

    // Check the postcondition.
    if (!(predicate code)) {
        constraintFailed("the constraint in OCL form");
    }
}
```

Around advice gives the aspect code access to a method's return value, this is necessary when evaluating a postcondition constraint that uses the result keyword.

The following aspect shows the required assertion code to enforce these three constraints:

- context Person inv: self.age >= 18
- context Person::implementRaise(raise:int) pre:
 self.salary + raise <= self.maxSalary
- context Person::implementRaise(raise:int) post:
 self.salary = self.salary@pre + raise

These constraints are placed upon a system consisting of a Person class which has an

attributes age, salary, and maxSalary all of type java.lang.Integer. The person class also has the method implementRaise(raise:int) used to add the raise to the person's salary.

```

privileged aspect Ocl2jAspect {
  private void inv0(Person self) {
    if (!((self.age.intValue()) >= (18))) {
      constraintFailed("self.age >= 18");
    }
  }

  // Before the execution of every public method in Person
  before(Person self):execution(public * Person.*(..)) && target(self){
    inv0(self);
  }

  // Around the execution of the implementRaise method in Person
  void before(Person self, int raise) : call(
    void Person.implementRaise(int))
    && target(self)
    && args(raise)
    && if (self.getClass() == Person.class) {
    if (!((self.salary.intValue() + raise) <=
      (self.maxSalary.intValue()))) {
      constraintFailed("self.salary + raise <= self.maxSalary");
    }
  }

  // Around the execution of the implementRaise method in Person
  void around(Person self, int raise) : call(
    void Person.implementRaise(int))
    && target(self)
    && args(raise)
    && if (self.getClass() == Person.class) {

    // Store the original salary.
    java.lang.Integer pre0 = self.salary;
    // Let the execution of the method proceed.
    proceed(self, raise);
    // Check the postcondition.
    if (!((self.salary.intValue()) == ((pre0.intValue()) + (raise)))) {
      constraintFailed("self.salary = self.salary@pre + raise");
    }
  }
}

```

```

// After the execution of every public method in Person
after(Person self) : call(public * Person.* (...)) && target(self) {
    inv0(self);
}

// After the execution of any constructor in Person
after(Person self) : execution(Person.new (...)) && target(self) {
    inv0(self);
}

void constraintFailed(String constraint) {
    // Logic to either throw an error or print the error to
    // standard error output stream.
}
}

```

Additionally, since the contract code is instrumented at the bytecode level, constraints can be set on classes for which source code does not exist. This is very advantageous in cases where one works with third-party libraries. For example, consider we're using a third party class for which we do not have the source code. Even though we do not have the source code we can specify partial¹⁵ pre- and postconditions on this class' methods. In this case the precondition will ensure that our application uses the component's methods correctly and the postcondition ensures that the class' methods return valid results.

A final advantage of using AspectJ for the purposes of instrumentation is that it allows the ocl2j framework to support the optimizations presented in Chapter 10. For example, consider that there is an invariant constraint on a very large collection, it takes a long time to check the constraint: `inv: arrayListCollection->includes(someObject),`

¹⁵ The constraints are partial as we are not familiar with the functional details of the class.

where `someObject` is at the end of a collection with millions of elements. Using AspectJ it is possible to determine when and how the collection has been modified by intercepting operations on the collection that can potentially modify this collection. This allows us to answer the following two questions:

- Has the collection changed?
- If yes, how has the collection changed?

With this information it can be determined whether the invariant has been violated without having to traverse the entire collection. This subject is explored in detail in Chapter 10.

7.2 `OclType::allInstances()` and `oclAny::oclIsNew()`

The implementation of these two operations requires special attention as it would be very difficult to provide an implementation for these two operations without the use of AOP as Java does not provide any functionality to which these operations could be mapped to.

`OclType::allInstances()` can be used on any type in the model (including user-defined types) and it returns the set of all instances of type and all its subtypes in existence at the time that the expression is evaluated.

When this operation is used on a type, say type `x`, a collection is added to the aspect. This collection contains all the references to that type `x` and of its subtypes. The references to objects of these types are stored as keys in a `java.util.WeakHashMap` residing in the aspect. An advice is added to the aspect to add the reference to an object of the type or the subtype after the execution of the constructor. The references are keys in the `java.util.WeakHashMap` as they will automatically be removed when the key is no longer in use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, which is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map. Note that if we did not employ the use of `java.util.WeakHashMap` to store the references of these objects then the garbage collection of these objects would never occur as the objects would be marked in-use (an object marked in-use cannot be garbage collected) by being elements of our collection. Conversely, an object is not marked in-use on account of being a key in the `java.util.WeakHashMap` data structure. The key to this discussion is that objects should be garbage collected if they are not used in the application code, even though they may be referenced by the aspect code.

`oclAny::oclIsNew()` can only be used in a postcondition (constraining a method) and evaluates to true if the object (instance) is created (using the `new` construct) during the execution of the constrained method, i.e., it didn't exist at precondition time. The

implementation for this operation is very similar to that of `OclType::allInstances()`, the major difference is that the newly created objects are stored in a collection only during the execution of the constrained method. The collection is emptied once the postcondition is checked.

The following is the aspect code for the constraint “context One::someOperation()

post: self.p.ocIsNew()” where self.p is an attribute of type Person.

```
privileged aspect Ocl2jAspect {
    java.util.WeakHashMap oclIsNewMap0;
    boolean oclIsNewCheckingOn0 = false;
    before(One self) : call(void One.someOperation())
        && target(self)
        && if (self.getClass() == One.class) {
        oclIsNewCheckingOn0 = true;
        oclIsNewMap0 = new java.util.WeakHashMap();
        }
    after() : execution(Person.new (...)) {
        if (oclIsNewCheckingOn0) {
            oclIsNewMap0.put(thisJoinPoint.getTarget(), null);
        }
    }
    void around(One self) : call(void One.someOperation())
        && target(self)
        && if (self.getClass() == One.class) {
        proceed(self);
        if (!(oclIsNewMap0.keySet().contains(self.p))) {
            constraintFailed("context One::someOperation() post:
                            self.p.ocIsNew()");
        }
        oclIsNewCheckingOn0 = false;
    }

    void constraintFailed(String constraint) {
        // Logic to either throw an error or print the error to
        // standard error output stream.
    }
}
```

The following is the aspect code for the constraint “context One inv:

self.p.allInstances()->size() > 0” where self.p is an attribute of type Person.

```
privileged aspect Ocl2jAspect {
    java.util.WeakHashMap
        allInstancesMap0 = new java.util.WeakHashMap();
    after() : execution(Person.new (...)) {
        allInstancesMap0.put(thisJoinPoint.getTarget(), null);
    }
    private void inv0(One self) {
        if (!((allInstancesMap0.keySet().size()) > (0))) {
            constraintFailed("context One inv:
                            self.p.allInstances()->size() > 0");
        }
    }
    before(One self) : call(public * One.* (...)) && target(self) {
        inv0(self);
    }

    after(One self) : call(public * One.* (...)) && target(self) {
        inv0(self);
    }
    after(One self) : execution(One.new (...)) && target(self) {
        inv0(self);
    }

    void constraintFailed(String constraint) {
        // Logic to either throw an error or print the error to
        // standard error output stream.
    }
}
```

7.3 Impact on Performance Due To AspectJ

Weaving the contract code with the program code at byte code level has numerous advantages as outlined in the previous sections, yet the issue of the cost that comes from using AspectJ as the weaving technology has still not been discussed. The creators of AspectJ claim that AspectJ’s performance should be on par with the same functionality

written without aspects. Any significant difference is a bug.

In the current version of ocl2j the impact on performance due to AspectJ can be deemed as acceptable since the impact on performance would be the same if the contract checking and enforcement code was inserted at the source code level.

The impact on performance due to the usage of AspectJ does become an issue when optimizations are implemented. Consider this constraint on a collection, `arrayListCollection->includes(someObject)`. To detect whether `arrayListCollection` includes the `someObject` element the following must be done for every method that can change the state of the collection:

- All operation invocations that can invalidate the constraint must be intercepted (for example, the `remove(object)` operation).
- A check must determine whether this operation is being invoked on the constrained collection (remember that the operation will be intercepted on every collection object as it is not possible to specify the collection object of interest directly).
- Data structures holding optimization related information must be updated (For example, in the case of the `arrayListCollection->includes(someObject)` operation this might be the number of times the `someObject` occurs in `arrayListCollection`).

This quickly amounts to a large performance penalty and will be the subject of future research. A detailed discussion on optimizations and the performance issues associated with them is found in Chapter 10.

8 THE OCL2J TOOL ARCHITECTURE

This chapter discusses the architecture of the ocl2j prototype, the tool developed in this work. This tool retrieves OCL constraints from the SUS model and uses the SUS source code to generate assertion code (the methods used to query the SUS model and source code are discussed in section 8.1.). The assertion code is later woven with the SUS code using the AspectJ compiler. During the execution of the instrumented SUS, the user is notified of any constraint violation.

The tool consists of two subsystems (shown in Figure 8-1): ocl2j and parser. Only the ocl2j subsystem is discussed in detail in this chapter (in section 8.2) as the (OCL) parser relevant information was given in Chapter 5 and further information can be found in [4].

Figure 8-1: ocl2j subsystem dependencies

The metrics showing the lines of code excluding comments (LOC) and the number of

classes in each package is shown in Table 11.

Package	Lines of Code	Number of Classes
ocl2j	3,949	27
parser	20,695	260
System Total	24,644	287

Table 11: ocl2j tool metrics

Upon constraint violation the user has a choice as to what happens when a constraint is violated (when an assertion fails):

- A runtime exception is thrown.
- A message is printed out to the system error output stream.

The thrown exception is a subclass of the `RuntimeException` class since runtime exceptions should be used to specify programming errors (as specified in [14]).

8.1 The Facades

This section describes the two facades present in the ocl2j tool that are used to extract the information needed in order to generate the assertion code for the SUS. Subsection 8.1.1 describes the model façade that extracts relevant information from a given SUS model and subsection 8.1.2 describes the reflection façade used to extract relevant information from the SUS source code.

The façade design pattern [34] provides a unified interface to a set of interfaces in a subsystem. A facade defines a higher-level interface that makes the subsystem easier to use.

8.1.1 The Model Facade

As the SUS model information can come from different sources (e.g., UML XMI file, metadata in the source code) this package contains an interface (`ModelFacade`) that each model façade class must implement. This interface (shown in Figure 8-2) defines operations to retrieve the OCL constraints (`hasNextConstraint()` and `nextConstraint()`) from the model along with information `ocl2j` requires for the constraint code generation that cannot be deduced from the SUS source code alone.

Figure 8-2: The `ModelFacade` Interface

The model façade is also used to extract the type of the elements in a collection and types of the keys/values in maps as Java 1.4 has no support for generics (also known as templates).

Finally, the `associationEnds:Set(String)` operation provides support for the OCL `OclType::associationEnds:Set(String)` operation that returns the set of names of the navigable `associationEnds` of a type, as they are defined in the model.

In this work the model information is acquired from the source code. The model information is embedded using Javadoc comments (via doclets). Detailed information on this topic can be found in appendix E.1. However, thanks to the use of the façade the solution can be easily extended to support other model sources, like a UML model stored in an XMI file.

8.1.2 The Source Code Facade

The Java Reflection API [40] is used to extract type information used in the constraints. The class `ocl2j::ReflectionFacade` acts as the façade for this information. The class is shown in Figure 8-3 followed by a description of the services it provides.

ReflectionFacade
<pre> +getClass(in name : String) : Class +getType(in className : String, in field : Class) : Class +getMethod(in className : String, in methodName : String, in parameters : Class[]) : Method +findMethod(in className : String, in methodName : String, in parameters : Class[]) : Method </pre>

Figure 8-3: The `ReflectionFacade` Class

The `getClass(name:String):Class` method returns an instance of the

`java.lang.Class` with details of the class with the name represented by the name string. The `getType(className:Class,field:String):Class` returns the type of the attribute field found in `className`. The method `getMethod(className:Class,methodName:String,parameters:Class[]):Method` returns the method found in `className` with the method name represented by `methodName` and the parameter types in the `parameters` array. The method `findMethod(className:Class,methodName:String,parameters:Class[]):Method` is similar to the method `getMethod`, but parameter types do not have to be specified exactly. The parameter types as described in Table 9 will be considered equivalent. Such a method is necessary as the actual parameter types used in query method within constraints may not be the ones that are specified in the source code. For example an add method that adds two integers passed into it might be defined in the source code with the parameters as being of type `java::lang::Integer`. Now, in the constraint the user write something like “`sum(self.someIntNumber, 1) > 0`”. We would first try to find a query method in the source code of with parameters of primitive type `int`, yet this would not be a fruitful search. Instead, a search is required that is aware of the fact that from OCL’s point of view a Java `int` type is equal to a Java `java::lang::Integer` type, thus bridging the OCL and Java types. `findMethod` is such a method.

8.2 The `ocl2j` package

This package contains two main classes: `Ocl2jGenerator` and `AspectGenerator`. The `Ocl2jGenerator` class traverses the abstract syntax tree, applies the transformation rules, and creates objects that represent each constraint. The aspect generator class (`AspectGenerator`) uses these objects to create the `Ocl2jAspect.java` file. The `Ocl2jAspect.java` file contains assertion code as well as supplemental static member classes.

There are three supplemental static member classes included in the `Ocl2jAspect` aspect: `OclCollection` provides collection operations that exist in OCL, but not in Java (see section 6.4.2), `OclInteger` provides the OCL-Integer `div` operation, and an instance of the `ConstraintError` class is thrown upon a constraint failure if the constraint enforcement is set to throw an error. `ConstraintError` is an unchecked exception as a broken constraint signifies a programming error and is a direct subclass `RuntimeException`. The Java assertion facility is not used to check the constraints as the user disables the checking of constraints by not weaving the program code with the `Ocl2jAspect`. Thus, the Java assertion facility provides no advantages.

9 CASE STUDY

The case study is based on the program presented in [6] – The “Royal and Loyal” System Example. This system was chosen due to the large number of diverse constraints being already defined for it, including some quite complex ones. Thus, using such a system shows the strengths and the weaknesses of the approach taken in this thesis. This chapter first introduces the SUS in section 9.1, subsequently metrics from scenarios being run on the original and instrumented versions of the SUS are shown and discussed in section 9.2, and the chapter ends with section 9.3 - the conclusions drawn from the case study.

9.1 The Royal and Loyal System

The program models a computer system for a fictional company called Royal and Loyal (R&L). R&L handles loyalty program for companies that offer their customers bonus points. Examples of companies that can take advantage of this program include a supermarket chain, a gas station chain, and an airline company. Customers receive a customer card to go along with their membership and their loyalty account. The membership has two levels: silver and gold. Faithful customers can get upgraded to the gold status which gives them access to a wider range of services.

Class	Description
LoyaltyProgram	Represents a loyalty program, in this case study the “Gold & Silver” program was used.
ProgramPartner	Represents a business in the program, for example, a supermarket chain.
Customer	Represents each customer.
Service	Represents the services available to customers.
ServiceSupermarketPurchase	The service offered by the supermarket chain where a purchase over \$25 is rewarded with 5 bonus points.
ServiceLevel	A level of service (silver or gold).
Membership	The membership each Customer object has with a LoyaltyProgram.
LoyaltyAccount	Represents the account for a customer in a LoyaltyProgram.
CustomerCard	Represents each the customer card for each customer.
Colour	Represents the colour of the CustomerCard, either silver or gold.
Transaction/Earning/Burning	The transaction a customer performs. If the bonus points are being gained then it’s an earning transaction. If the bonus points are being used then it’s a burning transaction.

Table 12: Royal and Loyalty class descriptions

Metric	Value
Number of Constraints	47
Program Bytecode Size	17,044 bytes
Lines of Code	381
Number of Attributes	53
Number of Classes	14
Number of Constructors	12
Number of Operation	34

Table 13: Royal and Loyal Program Metrics

9.2 Results

In this section two versions of the SUS are compared: the original (uninstrumented) version and the version of the SUS with the assertion code (the instrumented version). Initially, it was also planned to compare the version of the SUS instrumented by ocl2j against the version instrumented by the toolkit but unfortunately that was not possible due to major problems with the toolkit. For example, it could not handle constraints which access static attributes. Further, after removing those constraints the toolkit would generate an instrumented version of the SUS, but it was instrumented incorrectly as executing the instrumented SUS would result in errors being thrown (examples of these errors are found in appendix B.7).

The two versions of the SUS, the original (uninstrumented) version and the ocl2j-modified (instrumented) version, are compared with respect to the following three criteria:

- The bytecode size of the classes – section 9.2.1.
- The time it takes to execute the program under various conditions – section 9.2.2.
- The memory footprint of the program under various conditions – section 9.2.3.

Techniques for program execution time measurement and memory footprint measurement were acquired from [38].

Section 9.2.4 concludes by showing the errors thrown upon violation of an invariant, precondition, and a postcondition.

9.2.1 Bytecode Size

With respect to instrumentation, the increase in the size of the bytecode depends on several factors:

- The ratio between the number of methods and the size of the method bodies. A method with a relatively small body and a complicated contract will have a relatively large assertion code overhead. Conversely, a method with a large body and a simple

contract will have a lower assertion code overhead. For example, as seen in Table 14, the `Customer` class has ten constraints (two are complicated – may take a relatively long time to validate/invalidate) and undergoes an increase in 6,579 bytes (5 times the size of the uninstrumented version) while the `Service` class has four relatively simple constraints and increases roughly three times in size.

- The number of class invariants combined with the number of public methods in the class. Each invariant assertion code must be inserted before and after every public method's logic in the body, thus the larger the number-of-invariants/number-of-public-methods the larger the assertion code overhead. This is also relative to the complexity of the invariants, as the more complicated the constraint the larger the assertion code.

In this case, since we are dealing with a small program with a large number of constraints the increase in the bytecode was significant; the bytecode grew 2.55 times in size from 16,861 bytes to 43,097 bytes. This can be viewed as a worst-case scenario as rarely will programs have this kind of number-of-constraints (47) to program-size ratio (381 LOC).

Class Name	Original Bytecode Size (in bytes)	Instrumented Bytecode Size (in bytes)	Number of Constraints
Burning.class	533	533	0
Colour.class	760	1,848	2
Customer.class	1,658	8,237	10
CustomerCard.class	1,448	5,463	5
Earning.class	533	533	0
LoyaltyAccount.class	1,023	4,576	6
LoyaltyProgram.class	1,549	3,276	5
Membership.class	876	1,363	6
ProgramPartner.class	862	3,980	5
Service.class	987	3,272	4
ServiceLevel.class	995	4,379	4
ServiceSupermarketPurchase.class	900	900	0
SilverAndGoldDriver.class	4,105	4,105	0
Transaction.class	632	632	0
Total	16,861	43,097	47

Table 14: Class sizes in original and instrumented versions of the SUS

9.2.2 Execution Time

The executions times of the two versions of the SUS were compared under five conditions:

- Scenario 1: 1 customer is added to the system after which 1 purchases transaction (over \$25 so that the bonus point logic was involved) is performed for 1 customer.
- Scenario 2: 10 customers are added to the system after which 10 purchases over \$25 are performed for 1 customer.

- Scenario 3: 100 customers are added to the system after which 100 purchases over \$25 are performed for 1 customer.
- Scenario 4: 1 customer is added to the system after which 100 purchases over \$25 are performed for 1 customer.
- Scenario 5: 100 customers are added to the system after which 1 purchase over \$25 is performed for 1 customer.

The scenarios vary with respect to the number of customers being added to the system and the number of purchases being performed, as these two variables have the biggest impact on the execution time of instrumented system.

Table 15 summarizes the execution times. The results were obtained by querying the system time (using the `System.currentTimeMillis()` method) before and after the execution of the scenario and recording the difference.

Scenario		Execution Time for Original Code (in milliseconds)	Execution Time for Instrumented Code (in milliseconds)	X Slower Execution of the Instrumented Code
1	1 customer / 1 purchase	38	62	1.63
2	10 customers / 10 purchases	41	64	1.56
3	100 customers / 100 purchases	47	111	2.36
4	1 customer / 100 purchase	39	76	1.94
5	100 customers / 1 purchase	47	106	2.25

Table 15: Execution time comparison

In addition to adding one customer and making one purchase, scenario 1 shows the increase in execution time due to the system initialization (the setting up of the instance of the `LoyaltyProgram` class with program partners and services). From scenarios 2 and 3, we see that as the number of customers being added to the system and purchases being made increases the execution time sharply increases. Results from scenarios 3 and 5 indicate that the number of customers is the main factor that drives that increase. An examination of the constraints that must be validated when a new customer is added to the system reveals that the system the `LoyaltyProgram::enroll(c:Customer)` must be invoked. Associated with this method invocation are the checks of the following constraints:


```

context LoyaltyProgram
inv: self.serviceLevel->size() = 2
inv: self.partners->size() > 0
inv: self.partners->collect(pp:ProgramPartner|pp.deliveredServices)
    ->forall(s:Service|s.pointsEarned = 0 and s.pointsBurned = 0)
        implies self.membership->collect(m:Membership|
            m.loyaltyAccount)->isEmpty()
inv: self.customer->forall(c1:Customer, c2:Customer |
    c1<>c2 implies c1.name<>c2.name)
inv: self.customer->isUnique(c:Customer|c.name)
inv: self.customer->forall(c:Customer|c.age >= 18)

```

These invariants are checked twice, before and after the execution of `LoyaltyProgram::enroll(c:Customer)`. Additionally, the following pre- and postconditions associated with this method must be checked:

```

context LoyaltyProgram::enroll(c:Customer)
pre: not self.customer->includes(c)
post: self.customer@pre->including(c) = self.customer
post: self.membership->select(m:Membership|m.customer = c)
    ->forall(m:Membership|m.loyaltyAccount->notEmpty() and
        m.loyaltyAccount.points = 0 and
        m.loyaltyAccount.transactions->isEmpty())

```

In the method body of `LoyaltyProgram::enroll(c:Customer)`, `CustomerCard` and `LoyaltyAccount` objects are created, and the `ProgramPartner::newCustomer()` method is invoked. This leads to the checking of: the invariants on `CustomerCard` and `LoyaltyAccount` after the execution of their constructor, and the `ProgramPartner` invariants (checked twice - before and after the execution of the method) along with the pre- and postconditions of `ProgramPartner::newCustomer()`.

```

context CustomerCard
inv: self.validFrom.before(self.validThru)
inv: self.colour = Colour.SILVER or self.colour = Colour.GOLD
inv: self.printedName = self.owner.title.concat(self.owner.name)

```

```

context LoyaltyAccount
inv: self.points >= 0

context ProgramPartner
inv: self.name <> ''
inv: self.numberOfCustomers >= 0

context ProgramPartner::newCustomer()
post: self.numberOfCustomers =
      self.loyaltyProgram->collect(lp:LoyaltyProgram|
        lp.customer)->size()

```

After profiling the assertion code it appears that the degradations in performance are due to the constraints on collection operations. Consider the relatively simple constraint (a part of the LoyaltyProgram invariant):

```

context LoyaltyProgram inv: self.customer->isUnique(c:Customer|c.name)

```

Operation `collection->isUnique(expr : OclExpression) : Boolean` results in true if `expr` evaluates to a different value for each element in `collection`; otherwise, result is false. In the contracts above (R&L program), customers must have a unique name. To evaluate this constraint the constrained collection must be traversed, the result from applying `expr` on each element must be stored in a new collection (say `isUniqueCollection`). If the result of `expr` on an element of `someCollection` already exists in `isUniqueCollection` then we know the constraint is falsified. Checking this constraint is CPU intensive and being an invariant it must be checked before and after the execution of every public method in `LoyaltyProgram`. This problem is investigated further in Chapter 10 where possible solutions are also suggested.

Thus, programs that have large collections with many complicated constraints associated with these collections can expect degradation in performance of 2 to 3 times. Otherwise, the degradation in performance is relatively small; the execution speed is slowed down by roughly 50%.

9.2.3 Memory Footprint

The memory footprint of a program shows the total amount of memory the program uses and consists of the memory usage by objects, classes, threads, native data structures, and native code. The memory footprint of the two versions of the SUS was measured under the same five scenarios as described in 9.2.2. Table 16 summarizes the memory footprint. The footprint was measured using the Windows Task Manager at the end of the execution of the scenario (before the program terminated).

Scenario		Memory Footprint for the Original Code	Memory Footprint for the Instrumented Code	Percentage Increase in Memory Footprint
1	1 customer / 1 purchase	5,448K	5,720K	5.0% (272K)
2	10 customers / 10 purchases	5,460K	5,748K	5.3% (288K)
3	100 customers / 100 purchases	5,852K	5,960K	1.8% (108K)
4	1 customer / 100 purchase	5,444K	5,936K	9.0% (492K)
5	100 customers / 1 purchase	5,840K	5,964K	2.1% (124K)

Table 16: Memory footprint comparison

The results in Table 16 indicate that the memory footprint increases are not very large as they range from 2.12% to 9%. The increase in the memory footprint is caused by the assertion code that must be added and the additional objects that must be created. Given that the SUS is a small program with a large number of often complex constraints it is safe to assume that in most cases the percentage increase in the memory footprint size will not exceed 9%.

In scenario 3, the smaller-than-expected percentage increase (when comparing with scenarios 1 and 2) can be attributed to the Java automatic garbage collection. The checking of constraints may create temporary objects that are only used for the validation of the constraint (see the discussion on the OCL collection-operation `isUnique` in the

previous section) and are garbage collected before the memory footprint. This also explains the result for scenario 5. In the case of scenario 4 it seems that those temporary objects are not garbage collected. In order to have more accurate results commercial profilers may have to be used.

9.2.4 Assertion-Failure Scenarios

This section exhibits the behaviour of the instrumented SUS upon the falsification of an invariant, a precondition, and a postcondition. In all three cases an error is thrown telling the user the invariant that failed along with a stack trace showing where the invariant failed to help with the debugging. The failure of constraints was achieved by seeding faults in the SUS code: constraints to be violated were selected (one invariant, one precondition, and one postcondition), and faults were seeded to violate them. In all three examples below we show three things: the kind of constraint violated, a textual description of the violated constraint, and the error message printed on to the standard error output.

Invariant failure, cause: trying to enrol a customer under the age of 18. From the stack trace we see that the constraint failed during the instantiation of the Customer class in the method `SilverAndGoldDriver::addCustomers()`, this method was in turn called from the main method found in `SilverAndGoldDriver`.

```
java.lang.Error: The constraint 'context Customer inv: self.age >= 18' failed.  
  at ocl2j.local.ConstraintFailure.constraintFailed(ConstraintFailure.java:20)  
  at Ocl2jAspect.inv2(Ocl2jAspect.java:17)  
  at Ocl2jAspect.ajc$after$Ocl2jAspect$495a(Ocl2jAspect.java:369)  
  at Customer.<init>(Customer.java:15)  
  at SilverAndGoldDriver.addCustomers(SilverAndGoldDriver.java:74)  
  at SilverAndGoldDriver.main(SilverAndGoldDriver.java:106)
```

Exception in thread "main"

Precondition failure, cause: trying to enrol the same customer twice.

```
java.lang.Error: The constraint  
'context LoyaltyProgram::enroll(c:Customer) pre:  
  not self.customer->includes(c)' failed.  
  at ocl2j.local.ConstraintFailure.constraintFailed(ConstraintFailure.java:20)  
  at Ocl2jAspect.ajc$before$Ocl2jAspect$1aba(Ocl2jAspect.java:154)  
  at SilverAndGoldDriver.addCustomers(SilverAndGoldDriver.java:76)  
  at SilverAndGoldDriver.main(SilverAndGoldDriver.java:107)
```

Exception in thread "main"

Postcondition failure, cause: the points on the instance of `LoyaltyAccount` were not set to 0.

```
java.lang.Error: The constraint  
'context LoyaltyProgram::enroll(c:Customer) post:  
  self.membership->select(m:Membership|m.customer = c)->  
  forAll(m:Membership|m.loyaltyAccount->notEmpty()  
    and m.loyaltyAccount.points = 0 and  
    m.loyaltyAccount.transactions->isEmpty())' failed.  
  at ocl2j.local.ConstraintFailure.constraintFailed(ConstraintFailure.java:20)  
  at SilverAndGoldDriver.enroll_aroundBody11$advice(SilverAndGoldDriver.java:599)  
  at SilverAndGoldDriver.enroll_aroundBody12(SilverAndGoldDriver.java)  
  at SilverAndGoldDriver.enroll_aroundBody13$advice(SilverAndGoldDriver.java:581)  
  at SilverAndGoldDriver.addCustomers(SilverAndGoldDriver.java:66)  
  at SilverAndGoldDriver.main(SilverAndGoldDriver.java:103)
```

Exception in thread "main"

9.3 Conclusion

This chapter shows the use of the ocl2j tool on the implementation of the expanded version of Royal and Loyal system found in [6]. This is a small program (381 LOC) with a relatively large number of classes (14), operations & constructor (46), and most importantly constraints (47), often complicated.

Effects of the instrumentation were measured in three ways: bytecode size increase, execution time increase, and memory footprint increase. The execution time and the memory footprint increases were measured in five scenarios: adding one customer to the system and making one purchase, adding ten customers to the system and making ten purchases, adding 100 customers to the system and making 100 purchases, adding one customer to the system and making 100 purchases, and adding 100 customers to the system and making one purchase.

Though additional case studies should be considered to draw more general conclusions, the R&L system allows us to reasonably come to the following conclusions.

After the instrumentation the bytecode grew three times in size. This increase in size was attributed to the fact that this small program has a large number of operations with very small method bodies and a large number of (often complex) constraints. In a realistic application the increase of the bytecode is expected to be much smaller as this system

used an exuberant number of constraints.

The execution time of the instrumented SUS was shown to increase between 50% and 100% with respect to the execution time of the uninstrumented SUS for scenarios where large collections were not used. In scenarios where large collections are used the execution time can be substantially greater. This was attributed to the large number of complex constraints on these collections that had to be repeatedly checked. This is a limitation of this work; however, Chapter 10 proposes an extension to the current methodology that has the potential to overcome this limitation.

In all five scenarios the increases to the memory footprint were very reasonable, between 2.2% and 9%. Given the nature of this particular SUS, it is expect that most real applications will undergo an increase in the memory footprint closer to 5% and rarely over 9%.

Finally, effects of falsified assertions were shown on all three types of constraints: invariants, preconditions, and postconditions. In all cases an error is thrown showing the user the constraint that failed along with the location of the failure.

Considering the fact that this was a small program with a large number of complex constraints it would be reasonable to assume that the results displayed here can viewed as “worst-case” scenarios. Whether this tool proves to be a viable solution for most systems

remains to be demonstrated by further experiments. Systems with large collections and complex constraints upon those collections may suffer unreasonable increases in execution time (see Chapter 10).

10 POSSIBLE OPTIMIZATIONS

Before discussing how the current approach can be optimized, the general topic of optimization must be addressed. In [42] William A. Wulf the gives the following aphorism: “More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity”. It is not hard to find evidence of this, for example, the instrumentation approach of the toolkit tried to provide an optimized solution to the problem of instrumentation of OCL constraints in Java. Unfortunately, they have failed on several levels as their solution is neither correct (some invariants are not enforced properly) nor efficient (every attribute in every class in the system is duplicated) mostly thanks to their optimization approach (see section 3.6).

This work strived for an unoptimized correct and efficient approach with the potential for optimization (once the unoptimized solution is perfected). Now that a correct and unoptimized solution exists the approach can be optimized in a controlled manner. This cannot be done without the existence of an unoptimized solution since one cannot be sure that the optimized solution does in fact give better performance. This leads us to the question of how can the authors of the toolkit were sure that their solution provided better

performance than an unoptimized solution without comparing the two.

This chapter deals mainly with the limitation of this work identified during the case study, where the instrumented version of the SUS suffers from unreasonable increases in execution time caused by complex constraints on large collection in the SUS. First, it is shown how AspectJ's inherent properties can be leveraged to overcome this problem followed by a more general discussion of other possible optimizations and open issues that must be addressed.

More specifically, an introduction to the topic of optimizing constraints on collections is given in section 10.1. Section 10.2 discusses how the optimization technique modifies the assertion code. Section 10.3 discusses how changes to collections are observed. Section 10.4 discusses the difference between simple and derived collections. Section 10.5 describes optimization techniques for OCL collection operations that do not list `OclExpression` in the argument list (along with an explanation as to why such a distinction is necessary). Section 10.6 discusses the more complicated case of optimization techniques for OCL collection operations that do list `OclExpression` in the argument list. Section 10.7 presents the results of a prototypical implementation of the optimizations for the `collection->includes(object:OclAny):Boolean` operation. Section 10.8 discusses the issues related to the usage of query methods in constraints that are to be optimized, and section 10.9 proposes some more optimization ideas and

discusses open issues that must be addressed.

10.1 Introduction to OCL Collection-Operation Optimizations

The issue of constraints on collections using OCL collection-operations requires special and careful attention as this is the biggest source of performance overhead caused by the checking of constraints in a system with large collections. Thus, the constraints on collections must be checked as infrequently as possible, furthermore only parts of a collection that changed should be checked.

In general there are three methods of checking constraints on a collection:

1. Every time the constraint is checked the entire collection is traversed. This is the solution currently used in the ocl2j approach where it was discovered that this approach is infeasible if the SUS has complex constraints on large collections (see section 9.3).
2. The entire collection is only traversed when a change to the collection is detected. This is more efficient than the previous solution but is still very inefficient as a lot of collection operations can be optimized so that only a few elements need to be looked at to see if the constraint on the collection is still satisfied (as described next). The optimization system in the toolkit attempts this approach (unsuccessfully as it does

not detect all changes to collections, see section 3.5.3).

3. Only the parts of the collection that have changed are checked. This last solution is the most efficient one but also the most complex one. This is the solution pursued in this chapter. The solution requires looking at each collection type and each collection operation to optimize the mode in which a constraint is verified on the collection. For example, consider the constraint “context World inv: person->count(name = 'Joe') = 1” on Figure 10-1. As long as the elements being added and removed from the collection can be intercepted (and they can thanks to AspectJ) then it is trivial to determine if the constraint is satisfied or not by just looking at the added and removed elements of the collection, thus the changes to the collection.

Figure 10-1 World-Person Class Diagram

Before the optimization techniques can be discussed several related topics must be addressed:

- The impact these optimizations have on the format of assertion code (section 10.2).
- The types of changes collections can undergo and how AspectJ can be used to detect

these changes (section 10.3).

- The distinction between simple and derived collections. Some collection operations generate new collections, thus these intermediate collections must also be dealt with. This topic is addressed in section 10.4.

10.2 Modifying the Checking of Collection-Operations in Assertions

This section deals with the first step towards the optimization of constraint checking on collections: replacing the constraint on the collection with its result. Instead of performing the constraint checking operation on the collection every time the constraint is checked, the result of checking the constraint is replaced with a value that represents the result of the last operation, then this value holding the result is only updated if the collection is modified (prior to the constraint being checked). The following example demonstrates how this checking system works and why it's only applied to collection operations.

The constraint “context World inv: countryNum > 0” on the class diagram showed in Figure 10-1 is a trivial one that only depends on an integer. There is very little overhead in checking this contract before and after each necessary method. In this case it would be less efficient to observe this variable and only check the invariant when a modification to the variable is detected than to just blindly check the invariant. The

reason for this is that such a system would need to use a flag that is set when a modification to the variable is detected; this flag would be later checked to see if the invariant must be checked. A method that sets this variable a large number of times creates much greater overhead than the overhead created by simple a read of the variable before and after method executions.

But, if the contract changes to:

```
context World inv: countryNum > 0 and  
    person->forAll (p:Person|p.name <> '')
```

Then the overhead in checking this contract before and after each necessary method is tremendously larger if the World-Person multiplicity is large as the collection of `Person` elements would have to be constantly traversed and validated.

If this large collection undergoes small changes the inefficiency can be greatly reduced by replacing the `person->forAll (p:Person|p.name <> '')` part of the contract with a value that would reflect the result of this operation, a Boolean value `personOneAdam` in this case. This is possible as each chain¹⁶ of collection operations in a constraint can be reduced to a Boolean, an Integer, or an object. Now, this local Boolean `personOneAdam`

¹⁶ A chain of collection operations is defined as a set of consecutive collection operations, for example `set1->union(set2)->notEmpty()` is a chain of two collection operations that can be reduced to Boolean result.

is updated only when there is a change to the collection. How this change is detected and the local result updated will be explained in the next section. The result (`personOneAdam`) is stored in the aspect.

The logic can be summarized with the following steps:

- If `person->forAll(p:Person|name <> '')` ever ceases to be true (this is checked whenever the collection is modified) then `personOneAdam` is set to false to ensure that the next assertion check fails.
- If `person->forAll(p:Person|name <> '')` never ceases to be true, `personOneAdam` is left set to true, and the next assertion will not be affected by this part of the computation.

10.3 Detecting Changes

Depending on the collection operation (the reason for differentiating between different collection operations will be explained in detail in the next section), the following changes to a collection may be considered:

- A change in the number of elements the collection has (induced by elements being added or removed from a collection).

- A change to a property of an element in the collection. For example, Consider the constraint `“context World inv: person->forall(p:Person|name <> ‘’)”` with respect to Figure 10-1. The important thing to notice here is that this invariant can be falsified if an element (of type `Person`) in the collection `person` changes in such a way that its `name` attribute becomes equal to `''`. Thus, even though number of elements in the collection has not changed, the constraint can still be falsified.

AspectJ provides functionality for intercepting writes to attributes and invocations of methods. In case of collections used in constraints this functionality is exploited in four ways for change detection purposes:

1. If an attribute referencing a collection that participates in an invariant is changed (and ends up referencing a different collection), then we know we need to recheck the entire collection to see if it satisfies the constraint on the collection.
2. Operations used to modify collections can be intercepted to see how the collection is being changed (adding/removing elements). This gives us the opportunity to inspect the elements being added and removed. For example, assuming the `add(o:Object)` method is used to add object to a collection, using AspectJ we can intercept the invocation of this method on a type and have access to the object being added and the collection that it's being added to.

3. In some collection operations the element content is important and a change to a collection includes the change to an element within the collection. Consider the already discussed example with respect to Figure 10-1: context World inv: `person->forAll(name <> '')` where a change to an element in the person collection can falsify the constraint. Thus, the attributes of the elements used in the collection-operation expressions must be observed for change (in this case the `Person::name` attribute needs to be observed for change).
4. In some collection operations an object is specified in the arguments list. If this object changes then the collection must be rechecked. For example, context World inv: `self.person->includes(self.tallestPerson)` where `tallestPerson` is an attribute of type `Person` in the context class. If `self.tallestPerson` changes in value then the constraint may have been falsified. Thus, `self.tallestPerson` must be observed for change and the collection must be re-checked in the case that it does change.

10.4 Simple and Derived Collections

In this thesis, a collection is simple when it does not have to be computed from other collections; it is easily accessible and is stored in a class attribute.

A derived collection is a collection formed from another collection or collections, for example, when a bag is turned into a set.

For example, the set of chained collection-operations:

```
set->union(set2 : Set(T))->union(set3 : Set(T)) : Set(T)
```

Is transformed into:

```
resultSet1 = set->union(set2 : Set(T))  
resultSet2 = resultSet1->union(set3 : Set(T))
```

The resulting set (resultSet2 in this case) and the intermediate sets (resultSet1 in this case) are cached after they are first computed (stored in the aspect that contains the code for this contract) as otherwise these intermediate collections would have to be regenerated during every invariant check. During the execution of a method the modification to anyone of the collections (set, set2, and set3) is propagated to the intermediate collections (if necessary) and from the intermediate collection to the final collection. Thus, the intermediate sets are kept updated in real-time so that the final collection may be kept updated in real-time.

10.5 Common Operations Not Listing an `OclExpression` in the Argument List

OCL collection operations can be divided into two groups: operations that have the `OclExpression` in the parameter list and ones that do not. The collection operations are

divided into these two groups as they have to be treated in a different manner when it comes to optimization. This section handles the simpler case where the operation parameter list does not have the `OclExpression`, while the subsequent section deals with the more complicated case of operations listing the `OclExpression` in the parameter list. A change to the collection in this section refers to the addition or removal of an element whereas, when `OclExpression` is in the parameter list, we need to also consider changes to element properties and attributes. This case will be considered in the next section.

As explained in section 10.2, a chain of collection operations are reduced to the (non-collection) result they provide: a Boolean, Integer, or an object. This result is then stored locally in the aspect that contains the code for the contract the collection resides in, with the exception of the `size()`, `isEmpty()`, and `notEmpty()` operations, as will be explained in this section.

The common (to all collection types) OCL collection-operations not listing an `OclExpression` in the arguments list will now be discussed (operations with similar properties are grouped together).

```
collection->size() : Integer  
collection->isEmpty() : Boolean  
collection->notEmpty() : Boolean
```

These are the only operations that don't store the result in the aspect code (the result is an attribute in an aspect) because they are inexpensive operations in terms of the performance penalty, all others do store the result in the aspect code. These operations can be executed directly on the constrained collection.

```
collection->includes(object : OclAny) : Boolean  
collection->count(object : OclAny) : Integer  
collection->excludes(object : OclAny) : Boolean
```

In the case of these operations and the remaining operations in this subsection the result is kept updated in real-time in the aspect code. The result in this case is `numOfOccurrences::Integer` for `includes` and `count`, and `isExcluded::Boolean` for `excludes`. Keeping the results updated in real-time means that during the execution of the SUS, whenever there is a change to the collection that change is immediately intercepted and the result is updated accordingly. For example, consider the `collection->count(object:OclAny):Integer` operation, whenever an element is added to `collection` it is checked to see if it is equal to `object`, if so the result (`numOfOccurrences`) is incremented. The reverse is done when `object` is removed from `collection`.

If variable `collection` or `object` is set to point at a non-equal reference (using the Java assignment operator `=`) then the result must be recomputed from scratch, as in this case we're either dealing with either a new collection or a new object.

```
collection->includesAll(c2 : Collection(T)) : Boolean  
collection->excludesAll(c2 : Collection(T)) : Boolean
```

The result is kept updated in real-time and the result must be recomputed from scratch in the case that either `collection` (`collection` or `c2`) is set to point to a non-equal collection.

<code>collection->sum() : T</code>

<p>The result (<code>sum:Integer</code> in this case) is kept updated in real-time and the result must be recomputed from scratch in the case that the collection is set to point at a non-equal collection.</p>
--

10.6 Common Operations Listing an `OclExpression` in the Argument

List

As was discussed at the beginning of the previous section, OCL collection operations can be divided into two groups: operations that have the `OclExpression` in the parameter list and ones that do not. While the previous section discussed the latter case, this section discusses the former case of operations common to all collections in the `collection->operation(expr : OclExpression) : Result` format. In the previous section results of collection-operations were kept track of in real-time, in this section this is not a viable solution as the time when the constraint is checked on the elements of such a collection matters. For example, class invariants are assumed to be valid in on method entry and exit, but are not guaranteed to hold during method execution.

Consider the following invariant with respect to Figure 10-1:

```
context World inv: person->exists(p:Person|p.name='John' and p.age=18).
```

Now consider a method in the context class that does the following:

1. Adds an element to the collection where the attributes name and age are not set (name = '' and age = 0).
2. The name is then set to 'John'.
3. The age is set to 20.

Notice that if the above invariant was checked in real-time (right when the modification occurs) it would fail as soon as the element is added to the collection (in step 1 above).

Another consequence of the collection-operation being in the form `collection->operation(expr:OclExpression):Result` is that a modification of the properties on an element in the collection may falsify the constraint and therefore all modifications to relevant attributes pertaining to `expr` must be intercepted.

Thus, after a collection of interest is changed by the addition or removal of an element, the updating of the results of the operation on the collection must be done after the method finishes executing. This means that the changes to the collection must be kept

track of (stored in the aspect): elements added to the collection are stored in the “added elements delta (AED)” collection, and elements removed from the collection are stored in the “removed elements delta (RED)” collection. These changes are collected in real-time: any time an element is added to the collection it is also added to the AED, similarly every time an element is removed from the collection it is added to the RED.

Additionally, elements that undergo changes to attributes that are used in an expression must be collected (in our example the name and age attributes must be monitored for change in objects that reside in the collection). After the method executes those elements must also be rechecked to ensure the satisfaction of the constraint on the collection.

If `OclExpression` includes attributes (not on the collection iterator as these are already handled) that are modified then the collection may have to be rechecked in its entirety. If this kind of modification results in the collection having to be rechecked from scratch the modification is referred to as an “attribute modification”. Consider the following example with respect to Figure 10-1 where a change to `self.countryNum` may require the entire collection to be checked from scratch:

```
context World inv : self.person->forall(p:Person |
  if (self.countryNum < 5) then
    p.name.size() => 5
  else
    p.name.size() => 10
  endif
```

This invariant specifies that if there are fewer than 5 countries in world then the Person

elements in the `self.person` collection can have names (specified by the `name` attribute) with a minimum of five characters. If the number of countries grows beyond 4 then the names must have at least 10 characters. The `self.person` collection would have to be rechecked from scratch if `self.countryNum` changes from a value smaller than 5 to a value equal to or greater than 5 (this is an example of an attribute modification). The presence of query methods (in this case `size()` is the query method) in `OclExpression` will be discussed in section 10.8.

A couple of OCL collection operations listing an `OclExpression` in the arguments list will now be discussed individually in order to exemplify how such operations can be optimized.

collection->exists(expr : OclExpression) : Boolean

The elements (if there is more than one) satisfying the constraint are stored in a collection in the aspect, the result-collection.

Elements that satisfy `expr` are observed; if relevant properties (i.e., attributes used in the `OclExpression`) are changed on one of those elements then the `expr` is re-evaluated on that element to see if it should still reside in the result-collection.

Upon evaluation of the AED and RED collections: elements in the AED satisfying `expr` are added to the result-collection (if not already present) and elements found in both RED and the result-collection are removed from the result collection.

A result-collection with a size of zero equals to the `exists` operation returning false and a non-zero size equals to the `exists` operation returning true.

Upon an attribute modification the result-collection must be regenerated from scratch. This is also the case if variable `collection` is set to point at a non-equal reference (using the Java assignment operator `'='`).

`collection->forAll(expr : OclExpression) : Boolean`

All elements in the collection are observed; if relevant properties are changed on one of those elements then the `expr` is re-evaluated on that element to see if the change invalidated the `forAll` constraint on the collection.

Upon evaluation of the AED: elements in the AED are checked to see if they satisfy `expr`, if any of those elements do not satisfy `expr` then the `forAll` constraint on the collection evaluates to false.

Upon an attribute modification the result-collection must be regenerated from scratch. This is also the case if variable `collection` is set to point at a non-equal reference (using the Java assignment operator `'='`).

10.7 Preliminary Results

The `ocl2j` tool was modified to support the optimization strategy for the `collection->includes(object:OclAny):Boolean` collection operation. The effects of this optimization are measured by comparing two versions of an instrumented SUS: the optimized version and the unoptimized version. The code for both versions of the aspects is found in Appendix F.

The SUS consisted of a class with two attributes. Attribute `c` is a collection of type

`java.util.ArrayList` while attribute `crucialElement` specifies an element that must always reside in the collection specified by `c`.

The constraint on the SUS is:

```
context One inv: self.c->includes(self.crucialElement)
```

The instrumented versions of the SUS compared in terms of execution time for these two scenarios:

Scenario 1 A collection of 1000000 is used where the element specified by the `includes` operation is the **first** element in the collection. Then, 100000 elements are added to collection using a public `addElement` method. Since the method used to add the elements is public the invariant is checked before every method execution.

Scenario 2 A collection of 1000000 is used where the element specified by the `includes` operation is the **last** element in the collection. Then, 1000 elements are added to collection using a public `addElement` method. Since the method used to add the elements is public the invariant is checked before every method execution.

In the scenario the location of the searched-for (in this case specified by `self.crucialElement`) element in the collection is highlighted as this location

determines the performance of the unoptimized solution. In order to evaluate the constraint the logic for the unoptimized checking of the `includes` operation determines the result by iterating through the entire collection and on each element of the collection is compared to the searched-for element. The operation returns true as soon as it finds the first elements that matches the searched-for element, thus no-matter how large a collection is, if the element is at the beginning (location) of the collection the unoptimized solution will very quickly determine that element does in fact exist in the collection. On the other hand, if the element resides at the end of large collection the entire collection must be iterated over before a match is found.

As described in section 10.5, the logic for determining the presence of the searched-for element in the collection using the optimized strategy is: the result (`numOfOccurrences::Integer`) is kept updated in real-time. Keeping the results updated in real-time means that during the execution of the SUS, whenever there is a change to the collection that change is immediately intercepted and the result is updated accordingly.

The execution times of both versions of the SUS under these scenarios are presented in Table 17. They were measured in the manner described in section 9.2.2.

	Execution Time Using the Unoptimized Version (in milliseconds)	Execution Time Using the Optimized Version (in milliseconds)
Scenario 1	328	350
Scenario 2	78,359	less than 1

Table 17: Preliminary Optimization Results

Even though these results are very preliminary the following conclusions can be drawn from this experiment. Scenario 1 was designed to show the cost of the overhead that the optimization technique introduces and it showed the introduced overhead in negligible. Scenario 2 models the situation in which the limitation of the unoptimized approach is exposed; in this case it was shown that the optimized solution tremendously outperformed the unoptimized solution. From this preliminary study it can be seen that these optimization techniques hold much promise.

10.8 Support for Query Methods

The ocl2j solution to the query method problem (optimizing contracts that use query methods) relies on the fact that the majority of query methods have a well defined postcondition specifying exactly the attributes and navigations that the query method uses to obtain the result. Thus, the query method is simply replaced with the postcondition for that query method. It can be determined automatically that the postcondition specifies the result exactly when the `result` for the postcondition is specified using only the '='

operator.

Consider a query method `countPlusValue(Integer num)` with the following postcondition: `result = num + collection->count(someObject)`.

This query method is used in the following contract:

```
context SomeClass inv original: countPlusValue(10) >= 10
```

This invariant would be transformed to the following invariant:

```
context SomeClass inv new: ( 10 + collection->count(someObject) ) >= 10
```

As can be seen from this example the new invariant no longer has to take query methods into account.

In the rare case that the post condition does not define the result of the method exactly, the query method used in the contract will be executed every time the contract needs to be evaluated. If the query method is used in a postcondition with a `@pre` keyword then the query method is executed prior to the execution of the method and the result of the query is stored in a temporary attribute of the aspect code for the contract.

For an example of a postcondition constraint does not specify behaviour of a method in enough detail consider the case of a query method that performs a complicated statistical analysis on an array of integers, say `listOfInts`. The post condition for such a query

method may be that the result is in the range of 0 and 1, “post: result $\geq$ 0 and result $\leq$ 1”. It is clear that this post condition does not specify the attributes and navigations the method depends on.

10.9 Miscellaneous Optimizations & Open Issues

- The optimization of query method checking for query methods without well-specified postconditions (see section 10.8). The optimized solution could involve caching the result of the query method and having an invalid-result flag related to the cached result. The query method code would need to be statically parsed to see which attributes and navigations must be observed for change. Once a change to those attributes and navigations is detected the invalid-result flag would be set.
- Automatic determination of immutable classes can make the optimizations much more efficient, this is discussed in detail in the following subsection (section 10.9.1).
- In [13] it is specified that for `java::util::Set`, “Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element”. In essence, this means that when

an object resides in a Java set the attributes used by the `equals` operation to determine equality between two objects cannot change while the object is inside the set.

Specifically, the `equals` method is used to determine if two objects represent the same thing, this usually involves comparing attributes between the two objects. For example, the following is the `equals` method found in the `java::awt::Point` class:

```
public boolean equals(Object obj) {  
    if (obj instanceof Point) {  
        Point pt = (Point)obj;  
        return (x == pt.x) && (y == pt.y);  
    }  
    return super.equals(obj);  
}
```

Here it can be seen that attributes `x` and `y` are compared for equality. Static analysis of the `Point.java` source-code file can be used to extract this information from the `equals` method and this information can then be used to make the optimization code more efficient. For example, let us assume that we have a collection of type `Set` (`setOfPoints`) with elements of type `java::awt::Point`, furthermore we have the following constraint on the collection: `setOfPoints->forall(p:Point | p.x > 0 and p.y > 0)`. If the optimization strategy for the `forall` operation described in section 10.6 is followed the `x` and `y` attributes of `Point` would have to be observed for change at all times. But, since we know that this is a collection of type `Set`, and we know that the `equals` method uses the attributes `x` and `y` for comparative purposes,

then we also know that these two attributes cannot change while they reside in this collection (as the points in the set must remain unique). Hence, these attributes do not have to be observed for change.

10.9.1 Class Mutability

As explained in [14]: “An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object. The Java platform libraries contain many immutable classes, including `String`, the primitive wrapper classes (like `Integer`), and `BigInteger` and `BigDecimal`.”

Immutable classes are important with respect to this work as objects that are immutable need not be observed for changes. One of the places where this is useful is in case of elements in collections. If the elements are immutable this greatly simplifies the collection change detection mechanism. More generally, whenever an immutable class is used in the context of a constraint it does not need to be monitored for change.

A class is guaranteed to be immutable if it follows these five rules:

1. No methods are provided that modify the object.
2. No methods may be overridden.

3. All fields are final (constant and not being able changeable by assignment).
4. All fields are private.
5. Class has exclusive access to any mutable components (composite aggregation).

For a complete discussion on the topic of immutable classes see “Item 13: Favor immutability” in [14].

At first glance it may seem trivial to determine whether a class is immutable or not by simply checking if the class enforces the five rules outlined above. Even though the first four rules are trivial enough to check, the fifth rule is more difficult and is an active research topic. A tool that helps in detecting immutable classes was developed in an IBM research lab [17]. Unfortunately the tool is not available for public use.

A trivial substitute for the tool is a list of classes that are known to be immutable (this is the approach taken by ocl2j). This list contains classes known to be immutable in the standard Java class library. Additional immutable classes can be added to the list by the user of ocl2j.

A class does not have to be fully immutable for us to take advantage of immutability. If only specific attributes of a class are of interest and they are final and reference either a primitive type (example: `int`, `boolean`) or an immutable object then there is no need to observe these attributes for changes.

For example, consider the constraint:

`self.collection->forAll(p:Person|p.name <> '')` with respect to Figure 10-1, where the attribute `Person::name` is of type `String` and the modifiability is set to `final` (the attribute can only be set once in a constructor method of the object). According to the optimization technique described in section 10.6: “all elements in the collection are observed; if relevant properties are changed on one of those elements then the `expr` is re-evaluated on that element to see if the change invalidated the `forAll` constraint on the collection”. In this case the `name` attribute of objects of type `Person` is the relevant property, yet in this case it does not have to be observed for change for two reasons:

- The attribute is `final`, thus it can not reference a different `String` object that could potentially falsify the constraint.
- The string object is immutable.

11 CONCLUSIONS AND FUTURE WORK

This chapter is divided into two sections: summary & conclusions (section 11.1), and future work (section 11.2). The summary & conclusions section provides an overview of the work done in this thesis along with the results and limitations of the work. The future work section identifies areas that need to be researched further in order to address the limitations of this work and to continue advancing the technology.

11.1 Summary and Conclusions

Presented in this thesis is a methodology, supported by a prototype tool (ocl2j), to automatically transform OCL constraints into Java assertions and instrument these into the target program. This is accomplished with the following approach:

OCL constraints are automatically extracted from a model of the software under study. Each constraint is then converted into an abstract syntax tree using a parser generated by a compiler compiler. The abstract syntax tree is then traversed while transformation rules (from OCL to Java) are applied, resulting in the generation of assertion code. During the generation of this assertion code, the model and the source code of the system under

study are queried when necessary. Both are used to map OCL types to Java types. The assertion code is generated for every constraint and placed in an aspect file. The aspect file is then used to weave the assertion code into the system under study's bytecode. During the execution of the system under study, the user may specify that upon the falsification of a constraint either a runtime exception is thrown or an error message is printed to the standard error output stream.

The transformations for the generation of Java assertion code from OCL constraints have been derived in a systematic manner with the goal that upon instrumentation the generated assertion code be efficient and introduce a low memory overhead. This was largely achieved thanks to a technique where new objects are only created when absolutely necessary (i.e., the information required is not already available).

By employing aspect oriented programming technology the methodology helps overcome the problems of source code pollution, the ability to implement all OCL functionality, and it provides a framework that allows for improvements to the current approach. Source code pollution is circumvented as the Java assertion code is instrumented at the bytecode level instead of the source code level. Support for the OCL operations `OclType::allInstances()` and `oclAny::oclIsNew()` is provided, these operations would be very hard to implement if aspect oriented programming was not employed. Additionally, it was shown that the major limitation of the ocl2j approach, the significant

execution time increase on programs with complex constraints upon large collections, can be overcome using functionality found in AspectJ. The functionality allows for changes to collections to be monitored so that the entire collection does not have to be traversed in order to determine if some constraint upon this collection has been falsified.

The case study measured the effects of the assertion code instrumentations in three areas: bytecode size, execution time, and memory footprint. It was revealed that for the majority of the time the increase in all three areas was not excessive. Most importantly, for the majority of instrumented programs the execution speed will most likely increase between 50% and 100% while the memory footprint will increase by 2% to 11%. It was shown that increases to the three areas are dependent on several program properties, like the total program size, the number of constraints specified, and the complexity of the constraints. A limitation of the approach was identified as programs with complex constraints on large collections may suffer excessive execution time increases. Solutions for this limitation are proposed and experiments showed that the current methodology can be extended to overcome these limitations.

In conclusion, this work shows that in most cases transformation of OCL constraints to Java assertions can be done in an efficient and effective manner. The overhead introduced by the assertion code is acceptable in most systems as generally the instrumented version of the system is only used during development and testing. Furthermore, in the case

where the current solution fails to provide an efficient solution an expansion to the approach is proposed that addresses the problematic issue. When it comes to optimization M. A. Jackson's words [41] are as true today as they were when he wrote this in 1975,

“We follow two rules in the matter of optimization:

Rule 1. Don't do it.

Rule 2 (for experts only). Don't do it yet – that is, not until you have a perfectly clear and unoptimized solution.”

Thus, this thesis allows for a clear understanding of an unoptimized solution and allows for future efforts in this area to create an optimized solution and compare their solution to the one presented in this work.

It must also be noted that in the case of tools that transform information from a higher level of abstraction to one at a lower level (e.g., MDA), inefficiencies are addressed as the technology evolves in a stepwise manner, allowing people to work at a higher level of abstraction making them more productive. The same thing can be seen happening with tools that transform OCL constraints into the implementation language code, so that when this technology matures writing assertion code manually will be as often necessary as is writing programs using assembly language today.

11.2 Future Work

This section addresses issues that should be addressed in the continued research of this area:

- The proposed optimizations system should be expanded to use the proposed optimizations so that the major limitation of the current approach can be eliminated.
- Additional case studies should be performed in order to understand the impact of ocl2j instrumented assertion code better. For example, in this case study the bytecode grew three times in size after the instrumentation of the assertion code. It is currently believed that this is not representative of what will happen with real programs; this should be proved or disproved by further case studies.
- As is discussed in section 4.5, currently, the inheritance of constraints is only supported for invariants, this should be expanded to pre- and postconditions.
- ocl2j should be integrated with the toolkit's framework in order to take advantage of their semantic analysis package and the normalization package. This would make the tool more user-friendly.
- Integration of oc2j into an integrated development environment (like Eclipse) would make employment of the tool (and therefore OCL) even simpler.

- Upon detection of an assertion failure the displayed error message should specify more details as to the way the assertion failed, instead of just reporting that the assertion failed and the location of the failure. For example, values of variables associated with the constraint should be displayed. Such information (easily available within the aspect) will help the user with the debugging process.

Appendix A General

A.1 Term Definitions

assertion	A piece of Java code that evaluates to a result of (primitive) Boolean type.
classifier	A mechanism that describes behavioural and structural features. Classifiers include interfaces, classes, data-types, and components.
context	A view of a set of related modeling elements for a particular purpose, such as specifying an operation.
immutable	A class whose instances cannot be modified.
iterating methods	The term “iterating methods” denotes the collection properties <code>forAll</code> , <code>exists</code> , <code>select</code> , <code>reject</code> , <code>collect</code> , <code>isUnique</code> , <code>sortedBy</code> , and <code>iterate</code> .

iterator	A temporary used variable used in iteration: a sentinel.
nested collection	A collection where each element in the collection is another collection.
OCL operation	An operation on an OCL type variable.
operation	A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.
post fragment	Code that evaluates a constraint after a method finishes executing.
preparation fragment	Code that determines the value of @pre feature calls.
transfer fragment	Code necessary to make values determined in PREPARATION code fragments available in POST code fragments.

A.2 Acronym Definitions

AED	Added Elements Delta
AOP	Aspect Oriented Programming

AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CPU	Central Processing Unit
DbC	Design by Contract
EBNF	Extended Backus-Naur Form
LALR	Look Ahead LR
LOC	Lines of Code
LSP	Liskov Substitution Principle
OCL	Object Constraint Language
ocl2j	OCL 2(to) Java
OMG	Object Management Group
OO	Object-Oriented
RED	Removed Elements Delta
SUS	System Under Study

UML Unified Modeling Language

XMI XML Metadata Interchange

A.3 OCL Grammar Changes

This section describes the changes and assumptions the authors of the toolkit have made with respect to OCL 1.3. Assumptions they made that have been accepted by OCL 1.4 will not be described. This summarizes Chapter 2 of [2].

A.3.1 Use of Qualifiers

Qualifiers, written in brackets after the path name of a feature call, can express two different things.

- Qualifying use: A qualifier is used to give the qualifying value of a qualified association, as in “context A inv: b[23].featureB”.
- Navigational use: A qualifier is used to refine the navigation to association classes, for example “context A inv: c[end2].featureC”. While this navigational use is necessary only with recursive associations, it is legal for every navigation to an association class.

There is no way to distinguish these two sorts of qualifiers. There are even expressions where both uses of the qualifiers would be necessary at once, but this problem is restricted to such models that contain a recursive, qualified association that has an association class.

Now consider the expression `d.f[end3]` in the context of G. The qualifier could either be used navigationally, so the result type of the expression is `Set(F)`, or it could be used qualifying (selecting the association qualified by the value of `end3`, the attribute of G) and the result type is F.

They propose to find a different way to express the navigational use of qualifiers. The

implemented type checker supports only the qualifying use of qualifiers. OCL 1.4 has not addressed this problem.

A.3.2 Precedence of Relational Operators

The OCL specification defines a precedence of $<$, $>$, $<=$ and $>=$ over $=$ and $<>$. This is misleading, because the OCL grammar does not allow relational expression with more than one relational operator, so that expressions like $a > b = c < d$ are illegal anyway (this should be written as $(a > b) = (c < d)$).

This solution was rejected from OCL 1.4 as described in [8]: "Since the relational operators may be defined as names of operations they can be used for other types as well. The precedence might be needed in those cases. Rejected."

A.4 Person Tax Example with ocl2j

This AspectJ code generated by ocl2j is equivalent to the toolkit code shown in 3.5.1.

For the constraint:

```
context Person::getIncomeAfterTax(tax:float):float
  post: self.age = self.age@pre
```

Note: age is of type `java.lang.Integer`.

The following preparation code is generated:

```
java.lang.Integer pre0 = self.age;
```

And the following constraint checking code is generated:

```
((self.age.intValue()) == (pre0.intValue()))
```

Appendix B Dresden OCL Toolkit Details

B.1 Semantic Analysis Details

This section discusses the semantic analysis performed by the toolkit and is a continuation from section 3.2.

B.1.1 Type Checking

Type errors are the more obvious sort of semantic inconsistencies that can be found in correctly parsed OCL expressions. For type checking, the class diagram is necessary. This class diagram information has to be acquired from different sources depending on the deployment context of the OCL compiler (the toolkit supports the following sources: XMI class diagram files and metadata encoded in the SUS's Javadoc comments). An existing interface provisioned by the toolkit can be adapted to different model information sources. Type checking is implemented as a tree traversal that computes node types and asserts type conformance to the operators. It will fail if a type does not conform to an expected type.

B.1.2 Invariants on the AST

This section describes consistency rules on OCL constraints that can be formulated as OCL constraints on syntax trees. The classes generated by SableCC for the OCL grammar are used as the model context the constraints are evaluated in. This allows using the OCL compiler's Java code generator to generate code that checks for these constraints.

The invariants are found in the file `tudresden.oc1.codegen.JavaCodeGenerator` in the toolkit and look like this:

```
// unique variables
tree.requireInvariant(
    "context AExpression inv : letExpression -> forAll ( le | not self . parent . boundNames ->
includes ( le . name ) ) ");

tree.requireInvariant(
    "context AStandardDeclarator : not parent . parent . boundNames -> includes (
name.toString() )");

tree.requireInvariant(
    "context AIteratedDeclarator : let ppb = parent . parent . boundNames in not ( ppb ->
includes ( iterator.toString() ) or ppb -> includes( accumulator.toString() ) )");

// no multiple iterators
tree.requireInvariant(
    "context AStandardDeclarator inv : declaratorTail -> isEmpty");

// explicitly qualified names
tree.requireInvariant(
    "context ANamePathNameBegin inv : boundNames -> includes ( name )");

// iterators where possible
tree.requireInvariant(
    "context AFeatureCall inv : "
    + "let iteratingMethodNames : Set ( String ) = Set { 'collect', 'exists' , 'forAll' ,
'isUnique' , 'reject' , 'select' , 'sortedBy' } in "
    + "iteratingMethodNames -> includes ( self . pathName . toString ( ) ) implies "
    + "( self . featureCallParameters . declarator -> size = 1 "
    + "and self . featureCallParameters . declarator . oclType = AStandardDeclarator )");

tree.requireInvariant(
```

```
"context ANamePathNameBegin inv : defaultContext <> '' implies boundNames -> includes (
defaultContext )");
```

```
// named constraints
tree.requireInvariant("context AConstraintBody inv : name -> size = 1");
```

B.1.2.1 Expression context

A classifier context requires an invariant stereotype, an operation context a pre or post stereotype.

Offending examples:

```
context Person pre: age>=0
```

Error: A precondition cannot apply to a class.

```
context Person::getIncome():Real inv: result>=0
```

Error: A method cannot have an invariant.

B.1.2.2 Iterators

Only iterating methods can have iterators.

Offending examples:

```
context Person
inv: self.getIncome(c:Company|exprWithC)>0
```

Error: It does not make sense for `getIncome()` to specify an iterator.

B.1.2.3 Multiple iterators

Multiple iterators are allowed only for `forAll`.

B.1.2.4 Standard context

The parameter expression of an iterating method with multiple iterators must not make use of the default navigation context.

Offending example:

```
context Company inv:  
employees->forAll(i, j | firstName<>lastName)
```

Error: The iteration expression is ambiguous. Does this mean
`i.firstName<>j.lastName` or `j.firstName<>i.lastName`?

B.1.2.5 `self` as a variable name

Though it is not a keyword of the grammar, the word `self` plays a very special role in OCL. It must not be used as the name of a SUS operation parameter, a `let` variable or an iterator.

Offending example:

```
context Company inv: employees->forAll(self|self.age>15)
```

Error: `self` is used as an iterator.

B.1.2.6 Iterating method parameters

All iterating methods, except for `forAll`, must have exactly one parameter.

Offending example:

```
context Company inv:  
employees->exists(i, j | i.firstName == j.lastName)
```

Error: Two iterators are used in this `exists` operation: `i` and `j` while only one is allowed.

B.1.2.7 Time expressions

A time expression (`@pre`) is only possible in a postcondition. Similarly, the feature `oclIsNew` can only be used in postconditions.

Offending example:

```
context Person inv: age@pre <= age
```

B.2 Normalization Details

This section discusses the normalizations performed by the toolkit and is a continuation from section 3.3.

Normalization steps are divided into two broad categories: grammar reductions and type feature reductions. This section concentrates on such reductions that will make the code generation process easier. For this reason emphasis is put on conditions an AST must fulfill after normalization. Such a condition might be that all iterating methods have an explicit iterator. They can be formulated as OCL constraints on the AST's classes.

B.2.1 Grammar reduction

A grammar reduction is a normalization step that simplifies an OCL constraint in that the normalized constraint does not use the full range of grammar rules and alternatives. The language of normalized expressions is a subset of the OCL.

B.2.1.1 Explicit qualification of names

OCL allows for either the specification of a navigation context (`context Company inv: self.manager.isUnemployed = false`) or the use of the standard navigation context (`context Company inv: manager.isUnemployed = false`). The following simplification steps remove the second alternative by stating the standard navigation context explicitly.

Inserting self: The OCL keyword that refers to the instance of the constrained type, `self`, can often be left out in OCL.


```
"context Person inv: age>10" -> "context Person inv: self.age>10"
```

Inserting iterators: Iterating methods can be given with or without an iterator.

The parameter expression is evaluated in the context of the iterator type, even if the iterator is not explicitly given. Missing iterators are inserted in this normalization step, given that the iterator is named.

```
"context Company inv: self.employee->select(age > 50)->notEmpty" ->
```

```
"context Company inv: self.employee->select(p | p.age > 50)->notEmpty"
```

B.2.1.2 Naming

Naming constraints: OCL constraints can be given a name. If no name is given this normalization step inserts a generated name. This is required by the toolkit's Java code generator.

```
"context Person inv: ..." -> "context Person inv invariant1: ..."
```

Naming iterators: The insertion of the default navigation context (described previously) requires that iterating methods name their iterator. This iterator can be inserted automatically where it is missing.

```
"context Company inv:
self.employee->select(age > 50)->notEmpty" ->
"context Company inv:
self.employee->select(p | p.age > 50)->notEmpty"
```

B.2.1.3 Simplifying iterating methods shorthands

Solve multiple iterators: A special variant exists for `forAll` where multiple iterators are allowed. This can be normalized to multiple calls to the same property with one iterator each.

```
"context Company inv: employees->forAll(
e1, e2|e1<>e2 implies e1.name<>e2.name) "
->
"context Company inv:
employees->forAll(e1 |
self.employees->forAll(e2|e1.name<>e2.name implies e1.name<>e2.name) )"

```

Expand shorthand for collect: The collection property of OCL type `collect` can be shorthanded to a simple dot notation. Normalized OCL should contain the extended notation only.

```
"context Person inv: managedCompanies.employees->includes(self)" ->
"context Person inv:
managedCompanies->collect(employees)->includes(self) "

```

B.2.1.4 Type information

Inserting iterator and variable type: OCL allows naming the type of an iterator, but does not demand it. The same is true for type declarations in `let` expressions. This normalization step inserts such type information.

```
"...forAll(p|p.age>18)" -> "...forAll(p:Person|p.age>18)"
"... let a=age in ..." -> "... let a:Integer=age in ..."
```

B.2.1.5 Variable clarification

For code generators it is helpful if variable names are used only once in an expression. This can be achieved by traversing the AST, collecting all variable names and replacing all occurrences of a variable whose name was used before by a new variable with a generated, unique name.

B.2.1.6 OCL Type Feature Reductions¹⁷

Feature reductions are normalization steps that use knowledge about the OCL type system and find equivalent constraints by replacing occurrences of operations in a constraint with equivalent operations. For example, an occurrence of the binary operation “-” on Real values could be replaced by the binary “+” and the unary “-”.

Many properties of OCL types, both collection and basic types, are defined through other properties and could be normalized to these properties. Such transformations may easily result in less efficient implementations and should therefore be applied only if necessary.

This can be implemented in two ways: letting the compiler do the reduction or leaving the reduction up to the code generator. The toolkit chose the latter option as the

¹⁷ This is known as “type system reduction” in the toolkit, but we have changed the name as the original name is counterintuitive: there is no reduction of types.

usefulness of the type system reductions depends very much on the target language of the compiler. For the Java code generator that is part of their work a class library is available that implements the full OCL type system.

B.3 An Example of the SUS after Contract Insertion

Consider the following code:

```
/**
 *
 * @invariant x_0: x >= 0
 */
class Hi {

    public int x;
    /* {transient=false, volatile=false}*/

    public void incX() {
        x = x + 1;
    }

}
```

After the toolkit injects the contract code and the contract checking code the code looks like this (formatted to ease readability):

```
/**
 *
 * @invariant x_0: x >= 0
 */
class Hi {

    public int x;
    /* {transient=false, volatile=false}*/

    public void incX_wrappedbyocl_Hi() {
        x = x + 1;
    }
}
/**
A wrapper for injection. Generated automatically, DO NOT CHANGE!
@author ocl_injector
@see #incX_wrappedbyocl_Hi()
```

```

*/
public void incX() {
    if (tudresden.ocl.injection.ocl.lib.Invariant.checking_flag)
        incX_wrappedbyocl_Hi();
    else {
        checkForChangedFeatures();
        try {
            tudresden.ocl.injection.ocl.lib.Invariant.checking_flag = true;
            tudresden
                .ocl
                .injection
                .ocl
                .lib
                .Invariant
                .checkVacantInvariants();
        } finally {
            tudresden.ocl.injection.ocl.lib.Invariant.checking_flag = false;
        }
        incX_wrappedbyocl_Hi();
        checkForChangedFeatures();
        try {
            tudresden.ocl.injection.ocl.lib.Invariant.checking_flag = true;
            tudresden
                .ocl
                .injection
                .ocl
                .lib
                .Invariant
                .checkVacantInvariants();
        } finally {
            tudresden.ocl.injection.ocl.lib.Invariant.checking_flag = false;
        }
    }
}
}
/**
A default constructor for injection, replacing the automatically generated constructor.
Generated automatically, DO NOT CHANGE!
@author ocl_injector
*/
Hi() {
    try {
        tudresden.ocl.injection.ocl.lib.Invariant.checking_flag = true;
        tudresden.ocl.injection.ocl.lib.Invariant.checkVacantInvariants();
    } finally {
        tudresden.ocl.injection.ocl.lib.Invariant.checking_flag = false;
    }
}
/**
A backup for detecting modifications. Generated automatically, DO NOT CHANGE!
@author ocl_injector
@see #x
*/
private int x_oclbackup812374 = x;
/**
Contains observers for modifications of this feature. Generated automatically.
DO NOT CHANGE!
@author ocl_injector
@see #x
*/
public final java.util.HashSet x_oclobserveinvariants812374 =
    new java.util.HashSet();
/**
Checks object features, whether they have changed. Generated automatically,

```

```

        DO NOT CHANGE!
@author ocl_injector
*/
private void checkForChangedFeatures() {
    if (x != x_oclbackup812374) {
        x_oclbackup812374 = x;
        tudresden
        .ocl
        .injection
        .ocl
        .lib
        .Invariant
        .notifyObservingInvariants(
            x_oclobserveinvariants812374);
    }
}
}
}

```

B.4 Invariant Checking Problem

This program shows how the toolkit fails to enforce the invariant “context A inv: b.x

> 0” when main is invoked on A.

```

package inv;

/**
 * @invariant newConstraint_0: b.x > 0
 */
public class A {

    public B b = new B(1);

    public void changeBx(int x) {
        b.x = x;
    }

    public static void main(String[] args) {
        A a = new A();

        // This invalidates the invariant, but the
        // falsification of the invariant is not
        // detected.
        a.changeBx(0);
    }
}

package inv;

```

```

public class B {
    public int x;

    public B(int x) {
        this.x = x;
    }
}

```

B.5 Query Method Usage Problem

This program shows how the toolkit fails to enforce the invariant “context A inv: b.getSum() > 0” when main is invoked on A.

```

package qm;
/**
 * @invariant newConstraint_0: b . getSum ( ) > 0
 */
public class A {
    public B b = new B(1, 0);

    public void invalidate() {
        b.y = -5;
    }

    public void change(int y, int z) {
        b.y = y;
        b.z = z;
    }

    public static void main(String[] args) {
        A a = new A();

        // Each of the following two statements
        // invalidate the invariant, yet this is
        // not detected.
        a.invalidate();
        a.change(0, 0);
    }
}

```

```

package qm;

public class B {
    public int y, z;

    public B(int y, int z) {
        this.y = y;
        this.z = z;
    }

    public int getSum() {
        return y + z;
    }
}

```

B.6 Constraints on Collection Elements Problem

This program shows how the toolkit fails to enforce the invariant “self.myB->forAll

(a | a.x = 0)” when main is invoked on A.

```

package cols;
import java.util.Vector;

/**
 * @invariant newConstraint_0: self . myB -> forAll (a | a.x = 0 )
 */
class A {
    /**
     * @element-type B
     */
    public Vector myB = new Vector();

    public B addEle(int val) {
        B b = new B(val);
        myB.add(b);
        return b;
    }

    public void invalidateInvariant(B b) {
        b.x = -1;
    }
}

```



```

    public static void main(String[] args) {
        A a = new A();
        B b = a.addEle(1);

        // The next statement invalides the invariant,
        // this is not detected.
        a.invalidateInvariant(b);
    }
}

package cols;
class B {

    public int x;

    public B(int x) {
        this.x = x;
    }
}

```

B.7 Runtime Errors Thrown by a Toolkit-Instrumented Program

This is an example of the type of errors that were encountered when using the toolkit with non-trivial constraints. The toolkit support group was not able to resolve these problems.

```
java.lang.reflect.InvocationTargetException
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:324)
  at tudresden.ocl.injection.ocl.lib.Invariant.invoke(Invariant.java:106)
  at tudresden.ocl.injection.ocl.lib.Invariant.checkVacantInvariants(Invariant.java:144)
  at instrumented.Customer.getTitle(Customer.java:166)
  at instrumented.CustomerCard.<init>(CustomerCard.java:9)
  at instrumented.CustomerCard.<init>(CustomerCard.java:54)
  at instrumented.LoyaltyProgram.enroll(LoyaltyProgram.java:67)
  at instrumented.SilverAndGoldDriver.test1(SilverAndGoldDriver.java:39)
  at instrumented.SilverAndGoldDriver.main(SilverAndGoldDriver.java:46)
Caused by: java.lang.RuntimeException: error accessing feature
printedName_oclobservinginvariants812374 on object instrumented.CustomerCard@f9f9d8
  at tudresden.ocl.injection.ocl.lib.Invariant.addObserver(Invariant.java:68)
  at tudresden.ocl.injection.ocl.lib.Invariant.onField(Invariant.java:82)
  at tudresden.ocl.lib.OclAnyImpl.getFeatureQualified(OclAnyImpl.java:135)
  at tudresden.ocl.lib.OclAnyImpl.getFeature(OclAnyImpl.java:95)
  at instrumented.CustomerCard.zzzCheckOclInvariantMethod812374_tudOclInv0(
    CustomerCard.java:151)
  ... 12 more
java.lang.RuntimeException: java.lang.RuntimeException: error accessing feature
printedName_oclobservinginvariants812374 on object instrumented.CustomerCard@f9f9d8
  at tudresden.ocl.injection.ocl.lib.Invariant.invoke(Invariant.java:118)
  at tudresden.ocl.injection.ocl.lib.Invariant.checkVacantInvariants(Invariant.java:144)
  at instrumented.Customer.getTitle(Customer.java:166)
  at instrumented.CustomerCard.<init>(CustomerCard.java:9)
  at instrumented.CustomerCard.<init>(CustomerCard.java:54)
  at instrumented.LoyaltyProgram.enroll(LoyaltyProgram.java:67)
  at instrumented.SilverAndGoldDriver.test1(SilverAndGoldDriver.java:39)
  at instrumented.SilverAndGoldDriver.main(SilverAndGoldDriver.java:46)
```

Exception in thread "main"

Appendix C The OCL Grammar

This chapter shows the SableCC grammar use in this work and then discusses problem and changes to the grammar.

C.1 SableCC OCL Grammar

```
Package parser;
```

```
Helpers
```

```
  all = [0..127];
  lf = 10;
  cr = 13;
  uppercase = ['A'..'Z'];
  lowercase = ['a'..'z'];
  digit = ['0'..'9'];
  number = digit+;
  line_terminator = lf | cr | cr lf;
  input_character = [all - [cr + lf]];

  simple_escape_sequence = '\\' ' "' | '\\' " | '\\?' | '\\\\' |
    '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence |
    octal_escape_sequence |
    hexadecimal_escape_sequence;
  s_char = [all - [' ' + ['\\' + [lf + cr]]]] | escape_sequence;
  s_char_sequence = s_char+;
  h_set = 'Set';
  h_bag = 'Bag';
  h_sequence = 'Sequence';
  h_collection = 'Collection';
```

Tokens

```
comment = '--' [[all - 10] - 13]* [10 + 13]?;

// comment = '--' input_character* line_terminator?;

dot = '.';
arrow = '->';

not = 'not';

mult = '*';
div = '/';
plus = '+';
minus = '-';

context = 'context';
enum = 'enum';
t_pre = 'pre';
t_post = 'post';
t_inv = 'inv';

equal = '=';
n_equal = '<>';
lt = '<';
gt = '>';
lteq = '<=';
gteq = '>=';

and = 'and';
or = 'or';
xor = 'xor';
implies = 'implies';

l_par = '(';
r_par = ')';
l_bracket = '[';
r_bracket = ']';
l_brace = '{';
r_brace = '}';
semicolon = ';';

dcolon = '::';
colon = ':';
comma = ',';
channel = '#';
at = '@';
bar = '|';
```

```

ddot = '...';
apostroph = ''';

t_let = 'let';
t_in = 'in';

t_if = 'if';
t_then = 'then';
t_else = 'else';
endif = 'endif';

t_set = h_set;
t_bag = h_bag;
t_sequence = h_sequence;
t_collection = h_collection;

bool = 'true' | 'false';
simple_type_name = ( uppercase (lowercase | digit | uppercase | '_')*
) |
    h_set | h_bag | h_sequence | h_collection;
name = lowercase (lowercase | digit | uppercase | '_')*;

new_line = line_terminator;
int = number;
real = number '.' number;
blank = 9 | ' '*;
tab = 9;

string_lit = ''' s_char_sequence? ''';

```

Ignored Tokens

```

comment,
new_line,
blank,
tab;

```

Productions

```

// The context declaration code is now handled by ocl2j as this parser
// could not handle package pathnames in the specification
// of the context of an operation or the specification of the type
// of a parameter variable.

```

```

expression =
    let_expression* implies_expression ;

if_expression =
    t_if [if_branch]:expression t_then [then_branch]:expression t_else
[else_branch]:expression endif ;

```

```

// added by wdzidek
implies_expression =
    logical_expression implies_expression_tail* ;

// added by wdzidek
implies_expression_tail =
    implies logical_expression ;

logical_expression =
    equivalence_expression logical_expression_tail* ;

// added (SableCC)
logical_expression_tail =
    logical_operator equivalence_expression ;

equivalence_expression =
    relational_expression equivalence_expression_tail? ;

equivalence_expression_tail =
    equivalence_operator relational_expression ;

relational_expression =
    additive_expression relational_expression_tail? ;

// added (SableCC)
relational_expression_tail =
    relational_operator additive_expression ;

additive_expression =
    multiplicative_expression additive_expression_tail* ;

// added (SableCC)
additive_expression_tail =
    add_operator multiplicative_expression ;

multiplicative_expression =
    unary_expression multiplicative_expression_tail* ;

// added (SableCC)
multiplicative_expression_tail =
    multiply_operator unary_expression ;

unary_expression =
    {unary} unary_operator postfix_expression |
    {postfix} postfix_expression ;

postfix_expression =
    primary_expression postfix_expression_tail* ;

// added (SableCC)

```

```

postfix_expression_tail =
    postfix_expression_tail_begin feature_call ;

// added (SableCC)
postfix_expression_tail_begin =
    {dot} dot |
    {arrow} arrow ;

primary_expression =
    {lit_col} literal_collection |
    {literal} literal |
    {feature} path_name time_expression? qualifiers?
feature_call_parameters? |
    {parentheses} l_par expression r_par |
    {if} if_expression ;

feature_call_parameters =
    {empty} l_par r_par |
    {concrete} l_par expression fcp_helper* r_par |
    ( l_par declarator? actual_parameter_list? r_par ) ;

fcp_helper =
    {comma} comma expression |
    {colon} colon simple_type_specifier |
    {iterate} semicolon name colon simple_type_specifier equal
expression |
    {bar} bar expression ;

let_expression =
    t_let name let_expression_type_declaration? equal expression t_in
;

// added (SableCC)
let_expression_type_declaration =
    colon path_type_name ;

// added alternatives real and boolean
literal =
    {string} string_lit |
    {real} real |
    {integer} int |
    {boolean} bool |
    {enum} channel name ;

enumeration_type =
    enum l_brace channel name enumeration_type_tail* r_brace ;

// added (SableCC)
enumeration_type_tail =
    comma channel name ;

```

```

simple_type_specifier =
    {path} path_type_name |
    {enum} enumeration_type ;

literal_collection =
    collection_kind l_brace expression_list_or_range? r_brace ;

expression_list_or_range =
    expression expression_list_or_range_tail? ;

// added (SableCC)
expression_list_or_range_tail =
    {list} expression_list_tail+ |
    {range} ddot expression ;

// added (SableCC)
expression_list_tail =
    comma expression ;

feature_call =
    path_name time_expression? qualifiers? feature_call_parameters? ;

qualifiers =
    l_bracket actual_parameter_list r_bracket ;

// "iterate" alternative added (missing in specification)
declarator =
    {standard} name declarator_tail* declarator_type_declaration? bar
    |
    {iterate} [iterator]:name [iter_type]:declarator_type_declaration
semicolon
    [accumulator]:name [acc_type]:declarator_type_declaration equal
expression bar;

// added (SableCC)
declarator_tail =
    comma name ;

// added (SableCC)
declarator_type_declaration =
    colon simple_type_specifier ;

path_type_name =
    type_name path_type_name_tail* ;

// added (SableCC)
path_type_name_tail =
    dcolon type_name ;

```



```

// added to solve keyword conflict
type_name =
    {non_collection} simple_type_name |
    {collection} collection_type l_par simple_type_name r_par;

collection_type =
    {set} t_set |
    {bag} t_bag |
    {sequence} t_sequence |
    {collection} t_collection;

path_name =
    path_name_begin path_name_tail* ;

// added (SableCC)
path_name_begin =
    {type_name} type_name |
    {name} name ;

// added (SableCC)
path_name_tail =
    dcolon path_name_end ;

// added (SableCC)
path_name_end =
    {type_name} type_name |
    {name} name ;

// changed from "at name" to "at t_pre" to solve keyword conflict
time_expression =
    at t_pre ;

actual_parameter_list =
    expression actual_parameter_list_tail* ;

// added (SableCC)
actual_parameter_list_tail =
    comma expression ;

logical_operator =
    {and} and |
    {or} or |
    {xor} xor;

collection_kind =
    {set} t_set |
    {bag} t_bag |
    {sequence} t_sequence |
    {collection} t_collection ;

```

```

equivalence_operator =
    {equal}    equal |
    {n_equal}  n_equal ;

relational_operator =
    {gt}       gt |
    {lt}       lt |
    {gteq}     gteq |
    {lteq}     lteq ;

add_operator =
    {plus}     plus |
    {minus}    minus;

multiply_operator =
    {mult}     mult |
    {div}      div;

unary_operator =
    {minus}    minus |
    {not}      not;

```

C.2 OCL 1.3 Grammar Problems

In addition to the problems in the OCL 1.3 grammar that were found and fixed by the toolkit authors, the following problems were discovered:

- An incorrect context method parameter separation token was specified in the grammar, this has been fixed. This is taken into account by newer versions of OCL.
- The specification of the operation context does not allow for packages to be specified, this has been fixed. This is taken into account by newer versions of OCL.
- No escape sequences have been defined on the OCL String type, thus a string like

'Just call me 'James"' has undefined behaviour. This work assumes that OCL string will not contain inner OCL quotation character ('). This has not been taken into account in newer versions of OCL.

C.3 OCL 1.3 Grammar Changes

The only change to the OCL 1.3 grammar that makes a difference semantically is the requirement that operations with an empty parameter list use brackets. For example, the following syntax is not supported: `collection->size`. Since `size` is an operation it must be used in this manner: `collection->size()`. This change is in accordance with new versions of OCL.

Appendix D The Transformation Rules

For an explanation as to how to interpret this please see section 6.1. Each transformation rule is described in format presented in Table 3. Please note that some transformation rules are grouped together.

if_expression =
"if" [if_branch:] expression "then" [then_branch:] expression "else"
[else_branch:] expression "endif" ;
((ifBranchCode) ? (thenBranchCode) : (elseBranchCode))

expression =
logical_expression(1) implies_expression_tail* ;
implies_expression_tail =
"implies" logical_expression(2) ;
expression =
logical_expression(1) implies_expression_tail+ ;
boolean bImplies = logicalExpr1Code;
boolean b2Implies = logicalExpr2Code;
(!bImplies (bImplies && b2Implies))
The logicalExpression1Code & logicalExpression2Code is generated by another rule during the recursive top-down traversal of the AST.

<pre> logical_expression = equivalence_expression(1) logical_expression_tail* ; logical_expression_tail = logical_operator equivalence_expression(2) ; </pre>
<pre> logical_expression = equivalence_expression(1) logical_expression_tail+ ; </pre>
<pre> (equivalenceExpr1Code logical_operator equivalenceExpr2Code) </pre>
<p>The following transformation occur on the logical_operator:</p> <pre> OCL -> Java and -> && or -> xor -> ^ </pre>

<pre> equivalence_expression = relational_expression(1) equivalence_expression_tail? ; equivalence_expression_tail = equivalence_operator relational_expression(2) ; </pre>
<pre> (equivalence_operator = "=") and (equivalence_expression = relational_expression(1) equivalence_expression_tail;) and (relational_expression(1) & (2) are of the following primitive or reference type: Integer, Long, Short, Byte, Double, Float, Boolean) </pre>
<pre> relationalExpression1Code == relationalExpression2Code </pre>
<p>If either, relational_expression(1) or relational_expression(2) are of reference type then the primitive value is extracted with the according method invocation. For example, for the java::lang::Integer type the method used to extract the primitive value is intValue().</p>

<pre> equivalence_expression = relational_expression(1) equivalence_expression_tail? ; equivalence_expression_tail = equivalence_operator relational_expression(2) ; </pre>
<pre> (equivalence_operator = "=") and (equivalence_expression = relational_expression(1) equivalence_expression_tail;) and !(relational_expression(1) & (2) are of the following primitive or reference type: Integer, Long, Short, Byte, Double, Float, Boolean) </pre>
<pre> relationalExpression1Code.equals(relationalExpression2Code) </pre>

```

equivalence_expression =
    relational_expression(1) equivalence_expression_tail? ;
equivalence_expression_tail =
    equivalence_operator relational_expression(2) ;

```

```

(equivalence_operator = "<>") and
    (equivalence_expression =
        relational_expression(1) equivalence_expression_tail;)

```

```

relationalExpression1Code != relationalExpression2Code

```

```

relational_expression =
    additive_expression(1) relational_expression_tail? ;
relational_expression_tail =
    relational_operator additive_expression(2) ;

```

```

relational_expression = additive_expression(1)
    relational_expression_tail;

```

```

relationalExpression1Code relational_operator relationalExpression2Code

```

The tokens for the relational_operator are in OCL as they are equal to the ones in Java: <, <=, =, >=, >.

```

additive_expression =
    multiplicative_expression(1) additive_expression_tail* ;
additive_expression_tail =
    add_operator multiplicative_expression(2) ;

```

```

(additive_expression =
    multiplicative_expression(1) additive_expression_tail+;) and
    not ((add_operator = "-")
        and multiplicative_expression(1).isKindOf(Collection))

```

```

additiveExpression1Code add_operator additiveExpression2Code

```

The tokens for the add_operator are in OCL as they are equal to the ones in Java: + and -.

additive_expression = multiplicative_expression(1) additive_expression_tail* ; additive_expression_tail = add_operator multiplicative_expression(2) ;
(additive_expression = multiplicative_expression(1) additive_expression_tail+;) and ((add_operator = "-") and multiplicative_expression(1).isKindOf(Collection))
OclCollection.minus(multiplicativeExpression1Code, multiplicativeExpression2Code)
OclCollection is described in section 8.2.

multiplicative_expression = unary_expression(1) multiplicative_expression_tail* ; multiplicative_expression_tail = multiply_operator unary_expression(2) ;
multiplicative_expression = unary_expression(1) multiplicative_expression_tail+;
multiplicativeExpression1Code multiply_operator multiplicativeExpression2Code
The tokens for the multiply_operator are in OCL as they are equal to the ones in Java: * and /.

unary_expression = unary_operator postfix_expression ;
unary_operator postfixExpressionCode

unary_operator = "-" ;
-

unary_operator = not ;
!

<code>literal = string_lit ;</code>
<code>stringLiteral.replaceAll("'", "\\'")</code>
literal is used in <code>primary_expression</code> . The OCL string is encapsulated between two <code>`</code> characters. These characters are stripped and replaced with the java string encapsulation character: <code>"</code> . Section C.2 discusses the limitations with OCL strings.

<code>literal = real ;</code>
real
literal is used in <code>primary_expression</code> . Numbers are not transformed.

<code>literal = int ;</code>
int
literal is used in <code>primary_expression</code> . Numbers are not transformed.

<code>literal = bool ;</code>
bool
literal is used in <code>primary_expression</code> . OCL uses <code>true</code> and <code>false</code> to represent Boolean types the same way as Java, thus these are not transformed.

<pre> literal_collection = collection_kind "(" expression_list_or_range? ")" ; expression_list_or_range = expression(1) expression_list_or_range_tail? ; expression_list_or_range_tail = expression_list_tail+ ; expression_list_tail = comma expression ; </pre>
<pre> expression_list_or_range = expression(1); collectionType collection = new collectionType(); collection.add(expression1Code); for (int i=0; i < expressionListTail.size(); i++) { collection.add(expressionListCode[i]); } </pre>
collection
<p>literal_collection is used in primary_expression.</p> <p>Collection type transformations: OCL Bag & Sequence: collectionType = java.util.ArrayList OCL Set: collectionType = java.util.HashSet</p> <p>If every element in expressionListCode is a literal then this collection is only built once. Otherwise, this collection is rebuilt before every time (a constraint using this collection is checked).</p>

<pre> literal_collection = collection_kind "(" expression_list_or_range? ")" ; expression_list_or_range = expression(1) expression_list_or_range_tail? ; expression_list_or_range_tail = ddot expression ; expression_list_tail = comma expression(2) ; </pre>
<pre> literal_collection = collection_kind "(" expression_list_or_range ")" ; </pre>
<pre> collectionType collection = OclCollection.literalRange(collectionType, expression1Code, expression2Code); </pre>
collection
<p>literal_collection is used in primary_expression.</p> <p>The name of the collection variable is unique.</p> <p>Collection type transformations: OCL Bag & Sequence: collectionType = java.util.ArrayList OCL Set: collectionType = java.util.HashSet</p> <p>If expression1Code and expression2Code is a literal then this collection is only built once. Otherwise, this collection is rebuilt before every time (a constraint using this collection is checked).</p>

```

primary_expression =
  literal_collection |
  literal |
  "(" expression ")" |
  if_expression ;

```

For these cases the transformation have already been described.

```

primary_expression =
  path_name time_expression? qualifiers? feature_call_parameters? ;
primary_expression = path_name;

```

pathName

In this case path_name points to a field.
Every OCL package resolution "::" in path_name is changed to the Java package resolution "." in pathName.

```

primary_expression =
  path_name time_expression? qualifiers? feature_call_parameters? ;
(primary_expression = feature_call_parameters;) and
(the feature call list is empty)

```

pathName()

In this case path_name points to a method with an empty parameter list.

```

primary_expression =
  path_name time_expression? qualifiers? feature_call_parameters? ;
(primary_expression = feature_call_parameters;) and
(the feature call list is not empty)

```

pathName

In this case path_name points to a method with a non-empty parameter list. The parameter list will be built in the next stage.

primary_expression = path_name time_expression? qualifiers? feature_call_parameters? ;
(primary_expression = path_name time_expression;) and (not pathNameType.isKindOf(Collection))
pathNameType pre = pathName;
Pre
In this case path_name points to a field. This preparation code is executed before the constrained method executes.

primary_expression = path_name time_expression? qualifiers? feature_call_parameters? ;
(primary_expression = path_name time_expression;) and (pathNameType.isKindOf(Collection))
pathNameType pre = new pathNameType(); pre.addAll(pathName);
Pre
In this case path_name points to a field of type Collection. This preparation code is executed before the constrained method executes.

primary_expression = path_name time_expression? qualifiers? feature_call_parameters? ;
(primary_expression = path_name time_expression feature_call_parameters;) and (the feature call list is empty)
pathNameType pre = pathNameType();
pre
In this case path_name points to a method with an empty parameter list. This preparation code is executed before the constrained method executes.

```

primary_expression =
    path_name time_expression? qualifiers? feature_call_parameters? ;
(primary_expression =
    path_name time_expression feature_call_parameters;) and
    (the feature call list is not empty)
pathNameObjectType pre = pathNameObject.clone();
pre.pathNameFeatureCall
In this case path_name points to a method with a non-empty parameter
list.
This preparation code is executed before the constrained method
executes.

pathNameObject is the (object) owner of the method specified by
path_name and pathNameObjectType is its type. Section 6.5 discusses why
cloning is necessary here.

```

```

feature_call_parameters = "(" expression fcp_helper* ")" ;
feature_call_parameters = "()" ;

(expressionCode)
This is the parameter list for an operation with just one parameter.

```

```

feature_call_parameters = "(" expression(1) fcp_helper* ")" ;
fcp_helper = "," expression(X) ;
feature_call_parameters = "(" expression(1) fcp_helper+ ")" ;

(expressionCode1[, expressionCodeX]*)
This is the parameter list for an operation with more than one
parameter. The ", expressionCodeX" code is repeated * times.

```

```

unary_expression = postfix_expression ;
postfix_expression =
    primary_expression postfix_expression_tail* ;
postfix_expression_tail =
    postfix_expression_tail_begin feature_call ;

This is necessary factored out from all the subsequent
postfix_expression_tail_begin productions

```

postfix_expression_tail_begin = "." ;
feature_call =
path_name time_expression? qualifiers? feature_call_parameters? ;
For this production the previously described transformations apply.

postfix_expression_tail_begin = "->" ;
feature_call =
path_name time_expression? qualifiers? feature_call_parameters? ;
In the case of time_expression and qualifiers being present the
transformations described previously apply.

Appendix E ocl2j Usage Details

To create the `Ocl2jAspect.java` file invoke the tool execute `ocl2j.bat` from the classpath directory of the SUS. To instrument the bytecode of the SUS execute `ocl2jInstrument.bat` from the same directory.

The contents of the `ocl2j.bat` file describes how to the use of inheritance for invariants on or off, and how a constraint failure should be reported (an error is thrown versus an error is printed to the standard error output).

E.1 The Doclet Façade

Javadoc (doclet) comments use a very specific format:

```
/**  
 *  
 */
```

This format indicates to Javadoc that this is a Javadoc comment. Inside these comments the following metadata can be specified:

Doclet	Purpose
@pre	Use in the Javadoc comment before a method declaration to specify the precondition for that method. Example: <pre> /** * Add object the collection c. * @pre : o->notEmpty() */ public void add(Object o) { // Method code } </pre>
@post	Use in the Javadoc comment before a method declaration to specify the postcondition for that method. Example: <pre> /** * Add object the collection c. * @post : c->notEmpty() */ public void add(Object o) { // Method code } </pre>
@invariant	Use in the Javadoc before a class, field, and method declaration.
@element-type	Use in the Javadoc before a java.util.Collection field declaration. Example: <pre> /** * @element-type Person */ private Collection c = new HashSet(); </pre>
@key-type	Use in the Javadoc before a java.util.Map field declaration. Example: <pre> /** * Maps a person name to the object that * contains details on this person. * @element-type String * @key-type Person */ private Map map = new HashMap(); </pre>

Table 18: Usage Guide for the Doclet Facade

GUIDELINES

Do not specify the context of the class as this is retrieved automatically.

The pre/post/invariant doclets must be use in the following manner:

1. `@pre : constraint`
2. `@pre name_of_precondition: constraint`

Note the use of a space before the colon; it is not adjacent to the stereotype when no constraint name is specified.

Appendix F AspectJ Code for the Possible Optimizations

Chapter

This appendix shows the aspects for two versions of a constraint that uses the `collection->includes(object:OclAny):Boolean` collection-operation on a collection (see section 10.7 for details on the constraint). The first version of the aspect code (appendix F.1) enforces the constraint in an unoptimized manner while the second version of the aspect code (appendix F.2) enforces the constraint in an optimized manner with the technique discussed in section 10.5.

The optimized implementation is not complete as only the add and remove method are observed for the addition and removal of elements whereas a complete solution would also account for the fact object can be added and removed using the collection's iterator. The situation where the entire collection my rechecked due to collection or object being assigned a new value is not implemented either. The implementation of this functionality is not necessary in order to draw conclusions (from controlled experiments) regarding the execution times of two versions of the instrumented SUS: the optimized version and the unoptimized version.

F.1 Unoptimized Solution

```
import java.util.Collection;
import java.util.Iterator;

privileged aspect Ocl2jAspect {
  private void inv0(One self) {
    if (!(self.c.contains(self.curcialElement))) {
      throw new RuntimeException(
        "context One inv: self.c->includes(self.curcialElement)");
    }
  }
  before(One self) : call(public * One.* (...)) && target(self) {
    inv0(self);
  }

  after(One self) : call(public * One.* (...)) && target(self) {
    inv0(self);
  }
  after(One self) : execution(One.new (...)) && target(self) {
    inv0(self);
  }
}
```

F.2 Optimized Solution

```
import java.util.Iterator;

privileged aspect ExpAspect {
    java.util.ArrayList includesMap0 = new java.util.ArrayList();
    java.util.ArrayList includesMap0Object = new java.util.ArrayList();
    java.util.ArrayList includesMap0Num = new java.util.ArrayList();

    private void inv0(One self) {
        if (((Integer)includesMap0Num.get(indexOf(self.c))).intValue() <= 0) {
            throw new RuntimeException(
                "context One inv: self.c->includes(self.curcialElement)");
        }
    }

    before(One self) : call(public * One.* (...)) && target(self) {
        inv0(self);
    }

    // Logic required to keep track of the elements being removed to
    // collection of interest.
    after(java.util.Collection targetC, Object arg0) : call(
        boolean java.util.Collection.remove(Object))
        && target(targetC)
        && args(arg0) {

        int index = indexOf(targetC);
        if (index != -1) {
            Object objValue = includesMap0Object.get(index);
            Integer i = (Integer) includesMap0Num.get(index);

            if ((objValue == arg0) && (i.intValue() > 0)) {
                includesMap0Num.set(index, new Integer(i.intValue() - 1));
            }
        }
    }
}
```

```

// Logic required to keep track of the elements being added to
// collection of interest.
after(java.util.Collection targetC, Object arg0) : call(
    boolean java.util.Collection.add(Object))
    && target(targetC)
    && args(arg0) {

    int index = indexOf(targetC);
    if (index != -1) {
        Object objValue = includesMap0Object.get(index);
        Integer i = (Integer) includesMap0Num.get(index);

        if (objValue == arg0) {
            includesMap0Num.set(index, new Integer(i.intValue() + 1));
        }
    }
}

after(One self) : execution(One.new (...)) && target(self) {
    includesMap0.add(self.c);
    includesMap0Object.add((self.curcialElement));
    includesMap0Num.add(
        new Integer(
            ocl2j.local.OclCollection.count(self.c, self.curcialElement)));
}

after(One self) : call(public * One.* (...)) && target(self) {
    inv0(self);
}

after(One self) : execution(One.new (...)) && target(self) {
    inv0(self);
}

private int indexOf(Object o) {
    int num = 0;
    for (Iterator i = includesMap0.iterator(); i.hasNext();) {
        Object n = i.next();

        if (n == o) {
            return num;
        }

        num++;
    }
    return -1;
}
}

```

Appendix G References

1. T. Gjørøster, K. Haslum, Ø. Vinningland, “How to design and implement an OCL tool”, Agder University College, November 2002.
2. F. Finger, “Design and implementation of a modular OCL compiler”, Master Thesis Dresden University of Technology, March 2000.
3. R. Wiebicke, “Utility Support for Checking OCL Business Rules in Java Programs”, Diploma Thesis Dresden University of Technology, December 2000.
4. E. Gagmon, “SableCC, an object-oriented compiler framework”, Master Thesis McGill University, March 1998.
5. F. Finger, “Java-Implementierung der OCL-Basisbibliothek”, Großer Beleg, TU Dresden, 1999.
6. Object Constraint Language, Chapter 7 in [7].
7. OMG, “OMG Unified Modeling Language Specification”, Version 1.3, June 1999, <http://www.omg.org>.
8. OMG, “UML 1.4 RTF: OCL Issues”, Version 5, June 2000, <http://www.klasse.nl/ocl/ocl-issues.pdf>.
9. L. C. Briand, Y. Labiche, H. Sun, “Investigating the Use of Analysis Contracts to

- Improve the Testability of Object Oriented Code”, Software Practice and Experience (Wiley), vol. 33 (7), 2003.
10. The AspectJ Team, “Frequently Asked Questions about AspectJ”, June 2003,
<http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/faq.htm>.
 11. J. Warmer, A. Kleppe. “The Object Constraint Language: Precise Modeling with UML”, Addison-Wesley, 1999.
 12. Errata for [11]: <http://www.klasse.nl/english/boeken/errata.html>.
 13. Sun Microsystems, “Java 2 SDK Standard Edition Documentation”, Version 1.4.1.
 14. J. Bloch, “Effective Java: Programming Language Guide”, Sun Microsystems, 2001.
 15. OMG, “OMG Unified Modeling Language Specification”, Version 1.4, September 2001, <http://www.omg.org/technology/documents/formal/uml.htm>.
 16. J. Bloch, “The Java Tutorial, Trail: Collections”,
<http://java.sun.com/docs/books/tutorial/collections/>.
 17. S. Poraty M. Bibersteiny L. Kovedz B. Mendelsony, “Automatic Detection of Immutable Fields in Java”, November 2000,
<http://www.cas.ibm.com/archives/2000/proceed/cascon00/pdf/porat.pdf>.
 18. Elrad, T., R.E. Filman, and A. Bader, “Aspect-oriented programming: Introduction”, *Communications of the ACM*, 2001. 44(10): p. 29-32.
 19. Elrad, T., et al., “Discussing aspects of AOP”, *Communications of the ACM*, 2001, 44(10): p. 33-38.

20. Laddad, R., "I want my AOP! - Part 1", *JavaWorld Magazine*, January 2002.
21. The AspectJ Team, "AspectJ Quick Reference",
<http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/quick.pdf>.
22. Van Der Straeten, R. and Casanova Paez, M., "Stirred but not Shaken: Applying Constraints in Object-Oriented Systems. Proceedings", NetObjectDays, p. 138-150, ISBN/ISSN: 3000084193.
23. Sun Microsystems, Inc., "A Typesafe Enum Facility for the Java Programming Language", 2002,
<http://www.jcp.org/aboutJava/communityprocess/jsr/tiger/enum.html>.
24. IBM Rational Rose Website, <http://www-140.ibm.com/developerworks/rational/products/rose>.
25. Kleppe, A., Warmer, J., Bast, W., "MDA Explained: The Model Driven Architecture - Practice and Promise", Addison-Wesley, 1st edition, 2003.
26. Meyer, B., "Object-Oriented Software Construction", Prentice Hall, 2nd edition, 1997
27. Top 25 Java RFE's (Request For Enhancements), ID 4449383,
<http://developer.java.sun.com/developer/bugParade/top25rfes.html>.
28. JavaCC – The Java Parser Generator, <https://javacc.dev.java.net/>.
29. ANTLR Parser Generator, <http://www.antlr.org>.
30. W. Emmerich, "Engineering Distributed Objects", Wiley, 1st edition, 2000.
31. C. Szyperski, "Component Software", ACM Press, 2nd edition, 2002.

32. C. Ghezzi, D. Mandrioli, M. Jazayeri, "Software Engineering", Prentice Hall, 1st edition, 1991.
33. Eiffel Software Inc., "Building bug-free O-O software: An introduction to Design by Contract", <http://archive.eiffel.com/doc/manuals/technology/contract/>.
34. E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns", Addison-Wesley Pub Co, 1st edition, 1995.
35. B. Liskov, "Data Abstraction and Hierarchy", SIGPLAN Notices, 23(5), May, 1988.
36. R. Mitchell, J. McKim, "Design by Contract, by Example", Addison-Wesley Professional, 1st edition, 2001.
37. Cybernetic Intelligence GmbH, "OCL Compiler",
<http://www.cybernetic.org/prodoc115.htm>.
38. S. Wilson, J. Kesselman, "Java Platform Performance Strategies And Tactics", Addison-Wesley Pub Co, 1st edition, 2000.
39. Sun Microsystems, Inc., "Programming with Assertions (in Java)",
<http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>, 2002.
40. Sun Microsystems, Inc., "The Reflection API",
<http://java.sun.com/docs/books/tutorial/reflect/>, 2003.
41. M. A. Jackson, "Principles of Program Design", Academic Press, London, 1975.
42. W. A. Wulf, "A Case Against the GOTO", Proceedings of the 25th ACM National Conference 2 (1972): 791–797.

43. S. Cook, J. Daniels, "Designing Object Systems: Object-Oriented Modelling with Syntropy", Prentice Hall, 1994.
44. D. F. D'Souza, A. C. Wills, "Objects, Components, and Frameworks with UML : The Catalysis(SM) Approach", Addison-Wesley Pub Co, 1998.
45. K. Walden, J. Nerson, "Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems", Prentice Hall, 1995.