C# COMPILER EXTENSION TO SUPPORT THE OBJECT CONSTRAINT LANGUAGE VERSION 2.0

by

David Arnold

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Masters of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario 2004

Copyright © 2004 by David Arnold

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-97450-2

Our file Notre référence

ISBN: 0-612-97450-2

The author has granted a non-exclusive license allowing the Library and Archives 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 et Archives 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 thesis.

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

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

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

Canada

Abstract

The Object Management Group's Object Constraint Language (OCL), is part of the Unified Modeling Language (UML). The OCL is a formal language for the specification of behavioral constraints in software models. When models are transformed into executable code, such constraints are often lost or converted into non-executable comments. This research examines the integration of the OCL at the code level. Microsoft's .NET initiative introduces a new high-level language: C#. The C# language introduces several new concepts; including properties, delegates and events. The integration of the OCL into C# consists of creating not only a syntactical integration but, also a semantic one. Such integration will contribute to the verification of the OCL specification, as well as to provide a mechanism for the transformation of OCL expressions across the modeling levels defined under Model Driven Architecture (MDA).

Acknowledgements

Acknowledgements

I am grateful for the encouragement and feedback provided by my thesis supervisors: Jean-Pierre Coriveau and Francis Bordeleau. Without their help and support the thesis and implementation would not be what it is today.

A special thanks to Toby McClean and Juan Pablo Zamora Zapata. Toby pointed out errors and provided invaluable critiques of the implementation at various stages of its development. Juan Pablo showed me that Fritos and hot sauce are in fact breakfast foods.

Kevin Parker, my best friend, helped me to procrastinate at all hours of the day, with countless Halo games and movies. Thanks guy!!

I am indebted to Tony D'Ambrogio and Carleton University for the monetary support that was provided during the course of my research. I definitely would not have been able to focus on my research without their contributions.

Finally, my mother, my father and my sister have always been a source of encouragement, even when they had no idea what I was talking about.

Thank you all.

Contents

1	Introduction	1
1.1	Context and Motivation	1
1.1.1	Models	1
1.1.2	Design by Contract	2
1.2	Problem	3
1.3	Objectives	4
1.4	Contributions	4
1.4.1	MDA	4
1.4.2	OCL 2.0	5
1.4.3	Design by Contract	5
1.5	Thesis Structure	6
2	Background	8
2.1	Design by Contract	8

2.1.1	Elements of Design by Contract	8
2.1.2	Benefits of Design by Contract	13
2.1.3	Issues of Design by Contract	15
2.1.4	Summary of Design by Contract	19
2.2	The Object Constraint Language	19
2.2.1	OCL, UML, and MDA	20
2.2.2	PIMs and PSMs	20
2.2.3	Benefits of MDA	22
2.2.4	The UML and MDA	23
2.2.5	The OCL and the UML	23
2.2.6	Summary of the OCL, the UML, and MDA	32
2.3	Introduction to .NET	33
2.3.1	What is .NET?	33
2.3.2	The Common Language Runtime	34
2.3.3	Framework Class Library	37
2.3.4	Hands On .NET	38
2.3.5	Shared Source Common Language Infrastructure	41
2.3.6	The Mono Project	42
2.3.7	.NET Summary	42
2.4	C#	43

2.4.1	C# Overview	43
2.4.2	Types	44
2.4.3	Classes	46
2.4.4	Properties	47
2.4.5	Delegates	48
2.4.6	Events	50
2.4.7	Indexers	51
2.4.8	Operators	52
2.4.9	The <i>new</i> Modifier	53
2.4.10	Interfaces	54
2.4.11	Structures	55
2.4.12	Parameters	56
2.4.13	Attributes	58
2.4.14	C# and .NET	61
2.4.15	Summary	61
2.5	Existing Tools and Technologies	61
2.5.1	The Octopus Tool	62
2.5.2	The Object Constraint Language Environment	62
2.5.3	Spec#	63
2.5.4	Jcontract	64

2.5.5	iContract	64
2.5.6	Eiffel	65
2.5.7	Alloy	66
2.5.8	Summary	68
2.6	Summary	68
3	Syntax	70
3.1	OCLE Blocks	70
3.2	C# Grammar Changes	71
3.3	Parsing OCL Expressions	74
3.4	The OCL Grammar	75
3.4.1	Variable Expressions	75
3.4.2	Iterator Variable Lists	77
3.4.3	Iterators as Methods	77
3.4.4	Properties	78
3.4.5	Indexers	78
3.4.6	Summary	80
3.5	The Assignment of OCL Expressions	80
3.5.1	Association Context Declarations	80
3.5.2	Classifier Context Declarations	82
3.5.3	Operation Context Declarations	83
3.6	Summary	84

4	Semantics	86
4.1	Preconditions and Postconditions	86
4.2	Class Invariants	87
4.3	Constraints on Properties	89
4.4	Constraints on Indexers	89
4.5	Constraints and Delegates	89
4.6	Constraints and Events	91
4.7	Constraints and Inheritance	92
4.7.1	Class Invariants	92
4.7.2	Preconditions	94
4.7.3	Postconditions	95
4.7.4	Summary of Findler’s Method	96
4.8	Constraints on Interfaces	98
4.9	Constraints on Structures	98
4.10	Constraints on Abstract Classes	99
4.11	Constraints and The <i>new</i> Modifier	99
4.12	Constraints and the <i>override</i> Modifier	100
4.13	Summary	101

5	Resolution	103
5.1	Mono Compiler Preparation	103
5.1.1	Lexical Analyzer	104
5.1.2	The Parser	104
5.1.3	Parent Class Resolution	105
5.1.4	OCL Semantic Analysis	105
5.1.5	C# Semantic Analysis	105
5.1.6	Code Generation	105
5.1.7	Summary	106
5.2	The OCL Type Library Integration	106
5.2.1	Base Types	108
5.2.2	Primitive Types	112
5.2.3	Model Element Types	114
5.2.4	Collections	116
5.2.5	Iterators	121
5.2.6	Instances as Collections	121
5.2.7	Non-OCL Specific Elements	122
5.2.8	Handling Constraint Violation	123
5.2.9	Summary	125
5.3	Resolution of OCL Expressions	126

5.3.1	Constant OCL Expression Resolution	126
5.3.2	OCL Expression Resolution	134
5.4	Summary	166
6	Compilation	167
6.1	Mapping Init Rules to C# Elements	167
6.1.1	Fields	168
6.1.2	Constants	168
6.1.3	Summary	169
6.2	Mapping Derive Rules to C# Elements	169
6.3	Mapping Def Rules to C# Elements	172
6.4	Mapping Body Rules to C# Elements	172
6.5	Mapping Constraints to C# Elements	174
6.5.1	Resolution of Constraints	174
6.5.2	Value Saving	175
6.5.3	Class Invariants - Part 1	176
6.5.4	Preconditions	177
6.5.5	Method Path Modification	181
6.5.6	Postconditions	183
6.5.7	Class Invariants - Part 2	187

6.5.8	Summary	187
6.6	Code Generation	188
6.7	Optimization	188
6.8	Overall Compiler Flow: The Big Picture	189
6.9	Validation	190
7	Issues	192
7.1	OCL Issues	192
7.1.1	Character Sets	193
7.1.2	Null Values	193
7.1.3	Missing Collection Operators	194
7.1.4	Keywords	194
7.1.5	Ambiguous Grammar	196
7.2	Implementation Issues	197
7.2.1	Ambiguous Grammar	197
7.2.2	allInstances	197
7.2.3	Primitive Types	198
7.2.4	Properties and Indexers	199
7.3	Summary	199

8 Conclusion and Future Work	201
8.1 Contributions	201
8.2 Future Work	202
8.2.1 Implementation	202
8.2.2 OCL	203
8.3 Conclusion	204
Bibliography	205
A List of Acronyms	209
B Colour Plates	211
C C# Grammar to support the OCL	214
C.1 Lexical grammar	214
C.1.1 Line terminators	214
C.1.2 White space	215
C.1.3 Comments	215
C.1.4 Tokens	215
C.1.5 Unicode character escape sequences	216
C.1.6 Identifiers	216
C.1.7 Keywords	217

C.1.8	Literals	217
C.1.9	Operators and punctuators	219
C.1.10	Pre-processing directives	219
C.2	Syntactic grammar	221
C.2.1	Basic concepts	221
C.2.2	Types	221
C.2.3	Variables	222
C.2.4	Expressions	222
C.2.5	Statements	225
C.2.6	Namespaces	227
C.2.7	Classes	228
C.2.8	Structures	232
C.2.9	Arrays	233
C.2.10	Interfaces	234
C.2.11	Enums	234
C.2.12	Delegates	235
C.2.13	Attributes	235
C.2.14	OCL	236
C.3	Unsafe code	236

D	OCL Grammar	239
D.1	Grammar Header	239
D.2	Tokens	240
D.3	Punctuation	241
D.4	Precedence Rules	241
D.5	Context Declaration	242
D.6	Expressions	252
D.7	Grammar Footer	263
E	OCL Compiler Error Codes and Warnings	265
E.1	Errors	265
E.2	Warnings	276
F	Select Test Cases	278
F.1	Iterator Expressions	278
F.2	Navigation Expressions	284
F.3	Casting and State Expressions	286

List of Tables

2.1	Commonly used FCL namespaces	37
2.2	Common IL instructions	40
2.3	Summary of existing tools and technologies	68
5.1	Mapping between the OCL type library and C# types	108
5.2	Iterator operations defined on all collection types	136
6.1	Source code line distribution	190
7.1	The OCL keyword list	196
A.2	List of Acronyms	210

List of Figures

2.1	Suitable precondition for a pop operation defined on a stack	9
2.2	Postcondition illustrating the use of the <i>result</i> keyword	11
2.3	Method wrapped in preconditions and postconditions	11
2.4	Class invariant verifying the integrity of a stack	12
2.5	Method with preconditions, postconditions, and class invariants	12
2.6	Precondition to allow packages up to three kilograms	16
2.7	Precondition to allow packages up to five kilograms	16
2.8	Postcondition to ensure that packages must be delivered within five hours	16
2.9	Postcondition to ensure that packages must be delivered within two hours	16
2.10	Method with inherited preconditions and postconditions	17
2.11	Invariant ensuring at least one million dollars of insurance coverage	17
2.12	Invariant ensuring at least two million dollars of insurance coverage	17
2.13	Square root - defensive programming	18
2.14	Square root - design by contract	18

2.15 The MDA process	21
2.16 Example derivation rule to complete an incomplete UML model	24
2.17 Example initial value expression	25
2.18 Example initial value and derivation rule expression	25
2.19 Example body expression	26
2.20 Example attribute definition in the OCL	26
2.21 Example operation definition in the OCL	27
2.22 Example invariant	27
2.23 UML class diagram with a cycle	28
2.24 Class invariant to resolve the cycle	28
2.25 UML class diagram that contains dynamic multiplicity	29
2.26 Class invariant to resolve dynamic multiplicity	29
2.27 Class invariant to resolve optional multiplicity	30
2.28 Invariant placed on a state	31
2.29 The creation and execution of a managed .NET application	35
2.30 Squaring a number in IL (with line numbers)	39
2.31 Squaring a number in C#	41
2.32 Differentiating between value and reference types	45
2.33 Use of the object type	46
2.34 An abstract class	47

2.35 A sealed class	47
2.36 A Caption property defined on the Button class	47
2.37 Using the Caption property	48
2.38 Declaration of a new delegate type	48
2.39 Instantiation of a delegate type	49
2.40 Invocation of delegate instances	49
2.41 Illustration of a button click event	51
2.42 Firing a button click event (within the Button class)	51
2.43 Indexer with a single indexing parameter	52
2.44 Overloading of the + operator	52
2.45 Implicit and explicit type conversion operators	53
2.46 Use of the new modifier	54
2.47 Example interface	54
2.48 Explicit interface member implementation	55
2.49 Example structure definition	56
2.50 Use of reference parameters	57
2.51 Use of output parameters	57
2.52 Use of a parameter array	58
2.53 Code and metadata separation	59
2.54 Attribute declaration examples	59

2.55	Attribute definition example	60
2.56	Extraction of attribute information from metadata	61
2.57	Spec# syntax	63
2.58	Example of iContract syntax	65
2.59	A counter class written in Eiffel	66
2.60	Example contract specification using Alloy	67
3.1	Example of an OCL block with a single expression	71
3.2	Example of an OCL block with multiple expressions	71
3.3	C# grammar for parsing an OCL block	72
3.4	Placement of OCL blocks	72
3.5	Assignment of OCL blocks	73
3.6	UML class diagram showing the OCL grammatical elements	76
3.7	A modified iterator expression	77
3.8	OCL that allows constraints to be applied to C# properties	79
3.9	Use of the indexer keyword	79
3.10	Example of an association context declaration	81
3.11	An example of a classifier context declaration	82
3.12	An example of an operation context declaration	83
4.1	Method augmented with preconditions and postconditions	87

4.2	Method augmented with preconditions, postconditions and class invariants	88
4.3	Preconditions and postconditions used in the context of a delegate	90
4.4	Illustration of the add and remove event accessors	92
4.5	Inheritance of class invariants	93
4.6	Meyer's method for the inheritance of preconditions	94
4.7	Meyer's method for the inheritance of postconditions	96
4.8	Using a structure for interoperability between managed and unmanaged code	99
4.9	Precondition applied to an abstract method	99
4.10	Error case caused by the use of OCL and the <i>new</i> modifier	100
4.11	Use of the OCL with the <i>override</i> modifier	101
5.1	The OCL type library in a UML class diagram	107
5.2	Example use of the <i>OclVoid</i> type	110
5.3	Tuple values	110
5.4	Tuples all with the same value	111
5.5	Tuple type declaration	111
5.6	Tuple element access	111
5.7	OCL boolean expressions	112
5.8	OCL integer and real expressions	113
5.9	OCL string expressions	113

5.10 Use of collection types	117
5.11 Some operations defined on the Set type	119
5.12 Some operations defined on the OrderedSet type	120
5.13 OCL method call	121
5.14 OCL method call on a instance used as a collection	122
5.15 Usage of the Query attribute	123
5.16 Postcondition failure via an OCL exception	124
5.17 Tuple literal expression	127
5.18 C# code to represent a tuple literal expression	128
5.19 Collection literal expressions	128
5.20 C# code to represent collection literal expressions	128
5.21 Constant attribute call expression	129
5.22 Constant operation expressions	131
5.23 Resolved constant operation expressions	133
5.24 A let expression	134
5.25 C# code to represent a let expression	135
5.26 An if expression	135
5.27 C# code to represent an if expression	135
5.28 An iterator expression	136
5.29 Pseudo code for the resolution of the any iterator	138

5.30 Pseudo code for the resolution of the collect iterator	139
5.31 Pseudo code for the resolution of the exists iterator	140
5.32 Pseudo code for the resolution of the forAll iterator	141
5.33 Pseudo code for the resolution of the isUnique iterator	142
5.34 Pseudo code for the resolution of the one iterator	143
5.35 Pseudo code for the resolution of the reject iterator	144
5.36 Pseudo code for the resolution of the sortedBy iterator	146
5.37 An iterate expression	146
5.38 An iterate expression to calculate the sum of a set of integers	146
5.39 Pseudo code for the resolution of an iterate expression	148
5.40 A static attribute expression	149
5.41 Use of the collect iterator to ensure that each employee has a different social insurance number	151
5.42 Use of the shorthand collect notation to ensure that each employee has a different social insurance number	151
5.43 Use of the @pre keyword	152
5.44 Use of the -> operator	153
5.45 Resolution of the -> operator	153
5.46 Resolution of the oclIsTypeOf operation	155
5.47 Resolution of the oclIsKindOf operation	155

5.48	Pseudo code for the resolution of the <code>oclIsNew</code> operation	157
5.49	Use of the <code>oclIsInState</code> operation	157
5.50	Use of the <code>self</code> keyword	160
5.51	Use of the <code>result</code> keyword	160
5.52	Use of the <code>value</code> keyword	161
5.53	Use of a local variable expression	163
5.54	Promotion of a local variable for use in a local variable expression	164
5.55	Non-literal literal expressions	165
5.56	Resolution of non-literal literal expressions	166
6.1	IL code to initialize a field	168
6.2	Definition of a field named <code>_f1</code>	168
6.3	Definition of a constant in <i>C#</i>	169
6.4	Definition of a constant named <code>_c1</code>	169
6.5	A derive expression for the <code>_d</code> field	170
6.6	IL code for a derive rule	171
6.7	Body expresion for the multiplication of two numbers	173
6.8	IL code for a body expression	173
6.9	Truth table for processing preconditions	178
6.10	Precondition inheritance chain	179

6.11	IL code to implement a precondition inheritance chain	181
6.12	Method body before merging execution paths	182
6.13	Method body after merging execution paths	182
6.14	Truth table for processing postconditions	183
6.15	Postcondition inheritance chain	184
6.16	IL code to implement a postcondition inheritance chain	187
6.17	Test case to generate error OCL0085	190
7.1	Implementation of the equality operators on the Collection type	195
7.2	Using allInstances to state that a person may not have more than two parents	198
7.3	An OCL expression that states a person may not have more than two parents	198
B.1	OCL constraint failure dialog box	211
B.2	C#/OCL compiler interface	212
B.3	C#/OCL compiler structure	213

Chapter 1

Introduction

The introduction establishes the context and motivation by presenting a brief look at modeling and design by contract. The problem, namely adding constraints to a high-level language, is explicitly stated. The core objective of the research is presented. Several sub-objectives and the contributions are also presented. An overview of the remaining chapters completes the introduction.

1.1 Context and Motivation

The context and motivation for this research is driven from two different aspects of software engineering. The first is the use of models in the software development process. The second is the design by contract methodology. The following sections will examine each aspect in turn, and illustrate the relevance to this work.

1.1.1 Models

The introduction of the Object Management Group's (OMG)[36] model driven architecture (MDA)[9] has illustrated the importance of models. Software systems are consistently becoming more complex, and can no longer be handled at the code level. Models are needed to design a system at a higher

level of abstraction. MDA provides a mechanism for models, which are designed at this higher level of abstraction, to be automatically transformed into models that are more detailed. These more detailed models may even consist of code. Under MDA, code is considered a low-level model.

The process of a model transformation requires taking information specified at a higher level of abstraction and expressing it in a lower, or more detailed, one. Conventional modeling techniques consist of expressing system designs using the unified modeling language (UML)[33, 34]. UML models consist mostly of diagram elements. These diagram elements express a system at a high level of abstraction. During a model transformation to a lower level of abstraction, additional information is required. This additional information is not easily expressed with UML diagram elements. The additional information is often comprised of simple constraints and invariants. These are derived from the business rules for a system. The object constraint language (OCL)[32] is a formal language which acts as a plug-in to the UML. Such constraints and invariants can be expressed via the OCL.

With UML diagrams enriched using OCL constraints, the model transformations proposed under MDA can be achieved. Under MDA, the final transformation consists of transforming a detailed and implementation specific model into source code. At this level, the OCL constraints are transformed directly into source code statements.

The concept of MDA is a new one. Experienced developers are unlikely to switch from a source code driven method of design to a model driven one. The transition will occur in several steps[9, 45]. One of these steps is the ability to take existing source code and construct various models, representing a higher level of abstraction from the source code.

1.1.2 Design by Contract

Outside of the modeling world, a programming methodology known as design by contract[25, 29] uses constraints to express a system's design, and to enforce requirements. The Eiffel programming language directly implements the design by contract methodology[42]. The details of the Eiffel programming language, as well as other design by contract technologies, will be discussed in Chapter 2. Under

design by contract, constraints are expressed via three elements: preconditions, postconditions, and class invariants.

- Preconditions are placed at the beginning of a method to ensure that parameters passed to the method are valid.
- Postconditions are placed at the end of a method to ensure that the method executed correctly.
- Class invariants are associated with an attribute to ensure that the attribute maintains a valid state.

The constraints expressed by these elements can be expressed via a formal language such as the OCL. Having design by contract elements expressed using a different technology than that used for implementation creates a desirable separation. The importance of this separation will be seen shortly.

In order to compile a system which uses constraints, the constraints must be first transformed into the implementation language. The constraints are then tested at runtime to evaluate a system's integrity. If a constraint yields a value that does not satisfy the constraint, the constraint is said to fail. When a constraint fails, an exception is raised. Constraints defined under the OCL provide the syntax and semantics for generating such exceptions. Chapters 2 and 5 will discuss the action of raising an exception.

1.2 Problem

When the OCL, or any other constraint language, is transformed into source code, the idea of having specific constraints on a given program element needs to be retained. As programmers add functionality to a given method, the line between the code generated from the OCL and the code added by the programmer blurs. A clear separation between constraints and implementation would be achieved if a programmer could express constraints in a language different from the implementation.

With the exception of the Eiffel[42] programming language, there are no mainstream languages that support the specification of constraints. In addition, the Eiffel programming language does not separate constraints and implementation. Constraints drive the implementation. In order to increase the use of formal constraints at the code level, an extension to a popular programming language is required. Having

such an extension would provide benefits both at the source and modeling levels. Using constraints in each of the modeling levels aids in performing model validation, as the same formal constraints are present at each level. In addition, the model transformation process would be simplified if the constraints could be reproduced in the source code without the need for transformation to the implementation language.

1.3 Objectives

The main objective of this thesis is to create an extension to the C# programming language[11]. The rationale for selecting the C# language will be presented in Chapter 2. The extension will support formal constraints at the code level. Constraints will be expressed using version 2.0 of the OCL. In fulfilling the main objective, the following smaller objectives will be required:

- Construction of an OCL 2.0 parser.
- A C# implementation of the OCL 2.0 class library.
- Derivation of a mapping between each OCL construct and corresponding C# code.
- Development of a C# compiler that supports OCL expressions.

1.4 Contributions

By developing the C# compiler extension, several research contributions in the area of software engineering will be achieved. Specifically, this research will contribute to the disciplines of MDA, the OCL, and design by contract. The following sections will examine the contributions to each of these disciplines in detail.

1.4.1 MDA

The use of MDA relies on the transformation of models. For MDA to be used effectively, the model transformations need to be automated[9]. In order to automate a model transformation, the transfor-

mation action must be well defined. OCL expressions can be used in conjunction with UML diagrams at the highest level of abstraction. With support for the OCL at the code level, OCL expressions can simply be propagated¹ through the various model levels. A simple propagation operation results in a simple transformation step. A simple transformation will make it easy for developers of MDA tools to automate transformations.

1.4.2 OCL 2.0

The development of a complete implementation of the OCL 2.0 specification in C# will verify that the specification contains no ambiguities. At the time of writing, there are only two complete OCL 2.0 implementations. The first is the Octopus tool, implemented by Klasse Objecten[19]. The second is the LCI OCL Evaluator, implemented by Babes-Bolyai University[4]. Both tools are implemented using the Java language and operate at the UML level. The tools refer to the class diagram when evaluating OCL expressions. Neither of the tools makes use of an implementation level model. Existing implementations that do not have such a model cannot make use of these tools. The development of a code level implementation will allow systems that do not contain such models to make use of OCL constraints.

1.4.3 Design by Contract

Allowing programmers to express constraints on program elements directly using a formal language, adds to the benefits of design by contract. Constraints can be expressed separately from the implementation. Constraint support in a mainstream programming language facilitates the use of the design by contract methodology. Depending on the rules used in the compiler, the use of design by contract could even be enforced. Rules could be used to state that the compiler issues a compile-time error if a design by contract element is missing.

¹Without the need to be transformed.

1.5 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2 provides the necessary background information required for the understanding of the remaining chapters. Chapter 2 begins with an overview of design by contract. The notion of preconditions, postconditions, and class invariants are examined in detail. Several advantages to using the design by contract methodology are explored. Issues relating to the use of the methodology will round out the section. The OCL is explored by looking at the relationships between the OCL, the UML, and MDA. Following the exploration of the OCL, we will take an in-depth look at Microsoft's .NET technology. Particular attention is given to the .NET execution process and to managed code. Following our look into managed code, we will look at the details of Microsoft's .NET specific language: C#. The chapter finishes with a look into existing tool support for the OCL and design by contract. Particular interest is given to Alloy, Eiffel, iContract, Jcontract, the Octopus tool, and Spec#.

Chapter 3 explains how the OCL and C# are integrated at the syntax level. The notion of an OCL block is introduced, as well as the changes made to the C# grammar in order to incorporate the OCL. The chapter finishes by looking at the OCL parser and the grammar used to implement the parser.

Chapter 4 continues by explaining how the OCL and C# are integrated at the semantic level. Specific attention is given to the meaning of constraints on various C# program elements. The C# program elements examined include properties, delegates, and events. Chapter 4 looks at how inheritance is handled with respect to constraints. A modern inheritance method for contract inheritance proposed by Robert Findler[8] is examined in detail.

Chapter 5 begins the the action of compiling the OCL and C# together. The chapter starts by explaining the modifications needed so that an existing C# compiler can accept OCL expressions. Next, the integration of the OCL type library is explained. The chapter finishes by examining the OCL expression resolution process.

Chapter 6 completes the action of compiling the OCL and C# together. The chapter explores the mapping of the OCL init, derive, define, body, and constraint rules to their corresponding C# program

elements. The code generation process and the OCL specific compile-time optimizations are briefly discussed. The chapter concludes by looking at some of the tests used to verify the compiler.

Chapter 7 begins with discussing OCL specific issues that arose during the integration process. These issues include character sets, null values, and ambiguous grammar. Chapter 7 finishes by looking at the implementation issues that were discovered during the implementation process.

Chapter 8 provides a comprehensive summary of the work achieved. Finally, perspectives for further development and research are considered.

Appendix A provides a complete list of the acronyms used throughout this thesis.

Appendix B contains a collection of colour plates. The plates illustrate the implementation. They are referenced at various points in the thesis.

Appendix C lists the complete C# grammar that was used to parse C# integrated with OCL expressions.

Appendix D lists the complete unambiguous OCL grammar that was used to parse OCL expressions.

Appendix E provides a complete listing of all the OCL specific error and warning codes that the compiler can issue.

Appendix F contains a selection of the test cases used to validate the compiler. The complete set of test cases can be found in the implementation[2].

Chapter 2

Background

2.1 Design by Contract

Design by contract[25, 29] is one of many object-oriented design methodologies. As the name suggests, design by contract closely resembles a contract between two entities, such as a person and a business. When it comes to software development, design by contract represents a contract of sorts between two pieces of computer software. The design by contract methodology is not used for user to software relationships but, rather, software to software relationships[29]. The focus is on software-to-software relationships, as it provides benefits that affect the software development process. This section presents the three major aspects of design by contract: preconditions, postconditions and class invariants. The benefits of using design by contract will be explored. Issues relating to design by contract will also be examined.

2.1.1 Elements of Design by Contract

Design by contract is composed of three major elements. These elements are preconditions, postconditions, and class invariants. Each element will be examined below.

Preconditions

A precondition[25] is located at the beginning of a method. Preconditions specify requirements that must exist in order for a method to run correctly. If one or more requirements specified in a precondition fail, then an exception, in the C++ sense, is raised.

A precondition can be seen as a set of boolean conditions that must be true in order for a method to execute correctly. For example, consider a pop operation on a simple stack. A suitable precondition would be to check that the stack is not empty. If the pop method is called on an empty stack, the precondition fails and an exception is raised.

When a precondition fails, the caller of the method containing the precondition is responsible, not the method itself. The client (invoker) has violated its contract with the supplier (method).

In the OCL, a precondition is defined by a context declaration followed by the actual precondition. A context declaration is defined by the use of the *context* keyword, accompanied by the method declaration to which the precondition is being applied. Following the context declaration, the label *pre* is used to indicate the start of the precondition. Optionally, the name of the precondition can be written after the *pre* keyword in the label. Assigning a name to a constraint allows it to be referenced by name. Returning to our stack example, Figure 2.1¹ represents the precondition to check that the stack is not empty. The context declaration indicates that the precondition is being applied to the “Pop” operation defined on the “Stack” type. The use of the “OclAny” type at the end of the context declaration indicates that the “Pop” operation returns a value of type “OclAny”. More information on the OCL types will be provided in Chapter 5. The precondition is named “stackOk”. It states that the size of the stack must be greater than zero.

```
context Stack::Pop() : OclAny  
pre stackOk: stackSize > 0
```

Figure 2.1: Suitable precondition for a pop operation defined on a stack

¹In all OCL examples, the OCL keywords are shown in **bold** for readability.

Postconditions

A postcondition[25] is located at the end of a method. The postcondition ensures the executed method has completed all of its required tasks. If one or more of these tasks has not been completed, the postcondition fails and an exception is raised.

Like the precondition, a postcondition consists of a set of boolean conditions. All of the conditions must evaluate to true for the postcondition to be valid. The boolean conditions found in a given postcondition have two special features that are only available with postconditions.

The first special feature of postconditions is that they can reference the value of a given parameter, attribute or query method, before the method is executed. This feature can be used to determine if the value of a given parameter, attribute or query method has changed during the execution of the method to which the postcondition is assigned. Using our stack example, consider the operation of the pop method. The pop method takes the top element off the stack and returns it to the caller. A valid postcondition would be that the new size of the stack is one less than the size of the stack before the operation executes (Figure 2.2).

The second special feature is that postconditions can reference the return value of a method. This feature can be used to determine if the correct value is being returned to the method's caller. Returning to the stack example, if we define a query method called "Top", which returns the current element at the top of the stack, we can define a postcondition to make sure that we are returning the correct element (Figure 2.2).

In the OCL, a postcondition is defined by a context declaration followed by the actual postcondition. Following the context declaration, the label *post* is used to indicate the start of the postcondition. Like the precondition, a name may be specified after the *post* keyword in the label. The *@pre* modifier can be used after an attribute, a parameter or a query method name to indicate that the value before the method was executed is desired, rather than the current value. The *result* keyword is used to reference the return value of the method as illustrated in Figure 2.2.

When a postcondition fails in a given method, the fault lies in the method itself. The method (supplier)

```

context Stack::Pop() : OclAny
post stackPoppedOk: stackSize = stackSize@pre - 1
post stackPopReturnOk: result = Top@pre()

```

Figure 2.2: *Postcondition illustrating the use of the result keyword*

did not fulfill its part of the contract. The caller (client) did, as the precondition must have been valid. If the preconditions were invalid, an exception would have been raised before the method could execute.

With preconditions and postconditions together, we have a method prefixed with preconditions, and appended with postconditions (Figure 2.3).

```

myMethod(p1, p2)
{
    Preconditions
    Method Body
    Postconditions
}

```

Figure 2.3: *Method wrapped in preconditions and postconditions*

It should be noted, that if either preconditions or postconditions do not exist, they could be seen as a single condition, which is always true. In the context of contracts, a given method has a contract that binds it to its callers. If this contracted method is placed inside a class, then the class contains the following contract: “If you promise to call the method with its preconditions satisfied then I, in return, promise to deliver a final state² in which the postcondition is satisfied.”[29]

Class Invariants

Preconditions and postconditions are bound to individual methods. A class invariant[25] allows us to express constraints on the attributes for every instance of a class or structure. These constraints must be followed for every method contained within a class. In short, class invariants are used to capture integrity constraints and semantic properties which define a class[29].

With class invariants residing at the class level, it is important to know when the class invariant constraints are checked. Constraints are first checked when the creation method³ for an instance of a class

²The term “state” in the context of a class refers to the value of all of the data members.

³In most object-oriented languages, a creation method is analogous to a constructor.

has finished executing. If no creation method exists for a given class, a default one with an empty body is implicitly created. In this case, when a new instance of a class is created, the constraints are immediately checked. Following the creation of an instance, the invariant constraints are checked before a method is executed, and after a method has completed executing. It is important to note that a method may invalidate the constraints during its execution, but the constraints must be revalidated upon completion of the method. When an instance of a class is destroyed, a destruction method⁴ is called. In this case, the constraints are only checked at the beginning of the destruction method. Not having the constraints checked at the end of the destruction method allows the method to clean up any resources used, without having to worry about validating the constraints. Returning to the stack example, a corresponding class invariant would state that the value of the size attribute is always greater than or equal to zero (Figure 2.4).

```
context Stack
inv stackSizeOk: stackSize >= 0
```

Figure 2.4: Class invariant verifying the integrity of a stack

In the OCL, an invariant is defined by a context declaration, which consists of the *context* keyword, followed by the class or structure name to which the invariant is being applied. Following the context declaration, the label *inv* is used to indicate the start of the invariant. The name of the invariant may be written following the *inv* keyword in the label.

If we leave out instance creation and destruction, we can picture a class invariant as a set of constraints that are “and”ed to both the preconditions and postconditions of all methods in the class (Figure 2.5).

```
myMethod(p1, p2)
{
  Preconditions AND Class Invariants
  Method Body
  Postconditions AND Class Invariants
}
```

Figure 2.5: Method with preconditions, postconditions, and class invariants

⁴In most object-oriented languages, a destruction method is analogous to a destructor.

2.1.2 Benefits of Design by Contract

There are four main benefits to using design by contract[29]. These benefits are improved designs, improved reliability, improved documentation, and easier debugging.

Better Designs

Better designs can be achieved through the use of design by contract[29]. With programmers writing preconditions, postconditions, and class invariants, they are forced to think about the constraints. In having to think about the relationship between two programming elements, better designs can be achieved. If this thought process is used during the development of a single class, and the programming language enforces the uses of design by contract, the programmer will be forced to use a more systematic design process. As an example, if the implementation enforces the use of preconditions and postconditions, a developer is not likely to leave a method until the method's implementation adheres to all of the specified constraints. By using systematic design processes, better overall designs will be achieved[29].

Design by contract forces the explicit statement of rules that a client must follow. It also defines the actions provided by a supplier. When using program elements that were not written by a given programmer, it is often difficult to determine exactly what functionality a given element has. This raises some important questions when determining what functionality a programming element provides. With preconditions, postconditions, and class invariants located directly in a program element's code, answers to the questions can quickly be determined. Answers to these questions will make a programmer more likely to use the correct program element as they have access to information about the element.

Having the ability to specify the limitations of a given method via a precondition a programmer can, for example, avoid dealing with bad parameters. As such, the method does not have to contain "guard code", which deals with bad parameters. By not having to create "guard code", a programmer is likely to build classes with small, single purpose methods. Using small, single purpose methods allows for a better understanding of the design and possibly better reuse. As a small single, purpose method is easier to understand than a large complex one.

Exceptions are raised when a method is used incorrectly (false precondition), or does not execute correctly (false postcondition or class invariant). If we restrict the use of exceptions to this use only and not for the transferring of execution control, method bodies will not be cluttered. Less clutter means the code will be easier to read and understand. Some programmers use exceptions as a means to transfer execution control. Doing so creates a complex execution path, as exception raising statements generally transfer execution in a convoluted and complex way[29]. In order to prevent such complex execution paths, the use of exceptions is discouraged outside elements of design by contract.

Improved Reliability

Repetition allows for a better understanding of what is being done[29]. The same goes for design by contract. By creating contracts through preconditions and postconditions, a method's functionality is partially expressed. The actual code states how a method implements the functionality. Thinking about a method in two different ways increases the understandability of the method and reduces faults[29].

Writing contracts creates run-time assertions. Exceptions are raised if a contract is violated. Runtime checking of contracts allows bugs in the code to be found quickly. A bug will be discovered as soon as the application is run. If the application is written in a language which provides detailed exceptions, the failing assertion can be quickly located. The application state can also be determined by using the data that accompanies the exception.

Improved Documentation

Improved documentation is one benefit of design by contract[29]. When the contracts are written directly into the source code[42], as opposed to an external source[3], improved documentation results. Contracts are linked to a specific program element. Preconditions are linked to the beginning of a method; postconditions to the end; and class invariants to an attribute. This linkage allows the contract to become part of the public class view. The additional information provides the user with documentation indicating the restrictions on a given class, and an indication of what behavior resides within a class.

Contracts raise exceptions. Contracts can be used to enforce what the method actually does. If the contract changes, but the method does not, an exception will be raised at runtime. Conversely, if a method changes, but the contract does not, an exception will again be raised. The relationship between contracts and assertions ensures that the contracts are always in agreement with the code. If contracts are used to produce documentation, the documentation will also be in agreement with the code.

A tool can be used to extract contract information and place it directly into code level documentation. Such a tool is the XML comment generator which is found in the 2003 version of Visual Studio .NET[26].

Easier Debugging

As stated, contracts drive the generation of assertions. With runtime exceptions turned on, a bug can be located quickly. As contract checking decreases performance, it is normal for a programmer to check contracts during the development process only. In the case where performance is not mission critical, assertions may be left once development is complete. Customers can give developers information associated with an exception. The information allows a programmer to quickly locate the corresponding assertion, which in turn leads to easier debugging[29].

2.1.3 Issues of Design by Contract

Several issues exist when using design by contract[25, 29]. These issues pertain to inheritance, defensive programming, and using exceptions as control structures. We will examine each of these issues in the following sections.

Inheritance

In general, when a superclass contains contracts, all of its contracts are inherited by the subclass[29]. A subclass can refine the contracts provided by its superclass by restating them. All of the contracts in the superclass must be respected, as they become contracts in the subclass.

A precondition of a subclass can be less restrictive, but not more than the precondition defined in the superclass[29]. Consider a class that represents a courier service. The class has a single method named `deliver`, which takes as a parameter the weight of the package to deliver. Suppose a precondition on the `deliver` method states the weight of a package must be three kilograms or less (Figure 2.6).

```
context Courier::deliver(weight : Integer) : OclVoid
pre: weight <= 3
```

Figure 2.6: *Precondition to allow packages up to three kilograms*

If we inherit the “Courier” class to create a better courier service, and want to handle packages up to five kilograms, the precondition must be weakened (Figure 2.7).

```
context BetterCourier::deliver(weight : Integer) : OclVoid
pre: weight <= 5
```

Figure 2.7: *Precondition to allow packages up to five kilograms*

The net result of creating a new precondition for the `deliver` method is that the precondition of the superclass is “or”ed with the precondition of the subclass.

Postconditions in a subclass can be more restrictive than in the superclass, but not less[29]. Suppose the delivery method in the first courier has a postcondition that states the package must be delivered within five hours (Figure 2.8).

```
context Courier::deliver(weight : Integer) : OclVoid
post: time@pre + 5 > time
```

Figure 2.8: *Postcondition to ensure that packages must be delivered within five hours*

In our better courier service, we can tighten this statement. The package must be delivered within two hours (Figure 2.9).

```
context BetterCourier::deliver(weight : Integer) : OclVoid
post: time@pre + 2 > time
```

Figure 2.9: *Postcondition to ensure that packages must be delivered within two hours*

The net result is that the postcondition of the superclass is “and”ed with the postcondition of the superclass. Figure 2.10 presents a visualization of how preconditions and postconditions behave with respect to inheritance.

```

deliver(pl)
{
    Preconditions OR base.Preconditions
    Method Body
    Postconditions AND base.Postconditions
}

```

Figure 2.10: *Method with inherited preconditions and postconditions*

Class invariants are treated similarly to postconditions. The subclass inherits the invariants in the superclass, and can define new invariants. These invariants are “and”ed to the invariants of the superclass[29]. Suppose that the base courier service maintains an attribute indicating the amount of insurance coverage that the courier has. Suppose that we have an invariant on the attribute which states that we must have at least one million dollars of insurance coverage (Figure 2.11).

```

context Courier
inv: insurance >= 1000000

```

Figure 2.11: *Invariant ensuring at least one million dollars of insurance coverage*

The better courier company must have at least one million dollars of coverage (as the invariant was inherited). Increasing the restriction allows the better courier company to have more insurance coverage, but not less (Figure 2.12).

```

context BetterCourier
inv: insurance >= 2000000

```

Figure 2.12: *Invariant ensuring at least two million dollars of insurance coverage*

The main issue with inheritance of contracts is how blame is handled. The de-facto standard to handle contract inheritance is defined informally by Meyer[25]. Binder then formalized contract inheritance via the Percolation pattern[5]. The Percolation pattern provides an implementation for the automatic checking of superclass assertions to support design by contract. Recent work by Findler[8] has shown that Meyer’s and Binder’s method for the handling of blame is not sufficient. Meyer and Binder assign blame to either the client (invoker) or the server (supplier). However, they do not handle the case where the fault exists not with the client or the server, but rather with the inheritance chain. That is, where inheritance is being used incorrectly. In our implementation the method of contract inheritance proposed by Findler will be used. The details of which will be examined in Chapter 4.

Defensive Programming

The idea behind defensive programming is to try and handle all possible cases[25]. In the context of a method, this usually implies placing an “if” construct around each input parameter to verify its value. As an example, consider the case of a square root function that calculates the square root of only positive numbers. Figure 2.13 shows the defensive methodology. Here an “if” construct is used to ensure the parameter is positive. In design by contract we would replace this “if” construct with a precondition (Figure 2.14).

```
double MyClass::SquareRoot(int x)
{
    if(x < 0)
        throw new Exception("Negative value for SquareRoot");
    return sqrt(x);
}
```

Figure 2.13: *Square root - defensive programming*

```
context MyClass::SquareRoot(x : Integer) : Real
pre: x >= 0
```

Figure 2.14: *Square root - design by contract*

Defensive programming should not be used with design by contract. It is considered bad style to use both methodologies for the same constraint. Doing so doubles the work being done and makes the method more complex. Complex methods go against some of the benefits previously discussed. Most applications allow for both defensive programming and design by contract, as they each have a well defined role, but not in the same place. Design by contract is best used for software-to-software communication. The reason is that when a design by contract element fails it raises an exception. An exception is not desirable when a user enters a bad input value. Defensive programming would be a better choice, as the application should not fail when the user enters bad input values.

Exceptions as Control Structures

Exceptions should only be used to handle the failure of a contract, not as a control structure. Some programmers use exceptions to handle special cases, such as bad user input. In the context of design by contract, this is not only bad style, but against the methodology. A runtime exception is the indication of

a bug in the application, not a change in control. As previously stated, the use of exceptions as control structures creates a complex execution path. In order to reduce the complexity, and to only have a single meaning for the exception construct, exceptions should only be used for design by contract.

2.1.4 Summary of Design by Contract

Design by contract is an object-oriented design methodology made up of three elements: preconditions, postconditions, and class invariants. Preconditions must be true before a method is allowed to execute. Preconditions indicate rules that the client (invoker) must adhere to before using a method. Postconditions are tested at the end of a method's execution to determine if the method executed correctly. Postconditions ensure that the client gets the result that was advertised by the server (supplier). Class invariants are used to express constraints on global properties of a class or structure. Class invariants are tested when a new instance of a class or structure is created, as well as before and after the execution of every method. Class invariants are also tested before the destruction of an instance. Design by contract offers several benefits to the software development process. These benefits include improved designs, improved reliability, improved documentation, and easier debugging. Now our attention will focus on a formal language used to implement design by contract, the object constraint language.

2.2 The Object Constraint Language

The object constraint language (OCL) was first developed in 1995 during a business-modeling project within IBM. The OCL was designed to be both formal and simple. The syntax is very straightforward and can easily be picked up. In 1996, the Object Management Group (OMG)[36] was in the process of standardizing a language for object-oriented analysis and design. A proposal, of which the OCL was a fundamental aspect, was submitted by IBM and ObjecTime Limited. In November 1997, the OMG finalized the standard for object-oriented analysis and design. This standard was the unified modeling language (UML)[33, 34]. The OCL was included as part of the UML submission so that modelers are able to place information that cannot be expressed using UML diagrams into their models.

On September 18th 2000, the OMG issued a request for proposal (RFP) for a major revision to the OCL specification[31]. The main driver for issuing the OCL 2.0 RFP was the introduction of the UML 2.0 RFP. The main request was to construct an OCL metamodel. The metamodel allows for a clear separation between the OCL notation and its semantics. By adding the metamodel, the definition of a formal semantics, and a model interchange format (XMI), consistency across all OCL implementations would be possible. The OCL metamodel would place the OCL in alignment with the OMGs model driven architecture (MDA) initiative.

On August 8th 2003, the UML OCL 2.0⁵ draft adopted specification was issued by the OMG[32]. The document provides a high-level OCL description, an abstract syntax, a concrete syntax, a definition of the OCL standard library, and information on alignment between the OCL, UML, and MOF metamodels[35]. The following sections will look at some of the features found in the OCL including the relationships between the OCL, MDA, and the UML.

2.2.1 OCL, UML, and MDA

Model driven architecture (MDA) is becoming an important aspect of software development. MDA is a modeling framework being built within the OMG. It defines how models defined in one language can be transformed into models in a different language. One transformation is the mapping from a UML model to source code. Source code is considered a model under MDA[9].

2.2.2 PIMs and PSMs

The key to MDA is the importance of using models in the development process. Under MDA, the actual development process is driven by the modeling activity[45]. The MDA process consists of three main steps:

1. Construct a model with a high level of abstraction. The model will be free of all implementation technologies. This model is called a platform independent model (PIM).

⁵Unless specified, the term OCL is in reference to OCL 2.0 for the remainder of this thesis.

2. Transform the PIM model into one or more models that contain information, which is implementation specific. Implementation specific information can consist of data types and constructs that are available within a given implementation technology. Such a model is called a platform specific model (PSM).
3. Transform the PSM to implementation code.

In addition to the three steps already presented, the developer may wish to create a computation independent model (CIM). The CIM would be created before the PIM (Step 0). The CIM focuses on the environment of the system, and the requirements for the system. The model is oriented to the stakeholders of a system and hides all information relating to technology. As the OCL is a formal language which does not necessarily use syntax that is understood by the stakeholders, the OCL would not be introduced until the PIM level. As such, no further discussion of the CIM will take place.

As the PSM already contains implementation specific information, the transformation between the PSM and code is straightforward. The difficulty lies in transforming the PIM to the PSM. The relationships between the three steps is shown in Figure 2.15.

Figure 2.15: *The MDA process*

Under MDA, the transformations between the various models, PIM, PSM, and code are to be automated. The automation of these tasks will be a direct benefit of MDA. The transformation automation will remove the routine and mundane development tasks. An example of such a task is building a COM object from a high-level design. No matter what functionality the COM object entails, time will be

spent implementing the COM infrastructure. Under MDA, the infrastructure implementation would be left to the transformation tools.

It should be noted that the transformation tools are currently in their infancy, especially at the PIM to PSM level. With the formalization of the MDA standard, which is in agreement with the latest UML and OCL standards, it is expected that more interest and development of such transformation tools will occur in the near future[45].

2.2.3 Benefits of MDA

There are several benefits to MDA. Each benefit will be examined briefly below.

- **Portability:** MDA increases application reuse, and reduces the costs of application development and management. MDA allows for portability via platform independence. The platform independence is accomplished by the use of PIMs, which are truly platform independent. A PIM can then be used to generate several PSMs for different platforms.
- **Productivity:** The use of transformation tools allows, the generation of a PSM from a PIM, and the generation of code from a PSM to be fully automated. Automation ensures reliability and saves time. This provides a direct increase in productivity.
- **Cross-platform interoperability:** The use of tools to perform the transformations between models, ensures that standards based on multiple implementation technologies will be enforced to implement the identical business functions. The promise of cross-platform interoperability can be achieved by the transformation tools. The tools can also be used to construct bridges between different platforms.
- **Easier maintenance and documentation:** In MDA, most of the information about a given application is specified in a PIM. Under MDA, the action of building a PIM takes less effort than writing code. Less effort is required because most, if not all, of the implantation is to be generated via a model transformation tool.

2.2.4 The UML and MDA

In MDA, the unified modeling language (UML)[33, 34] can be used for specifying a PIM. With MDA, we want to be able to automatically generate PSMs. Following the PSM generation, we want to generate source code. In order to accomplish this goal, we will need a PIM rich in terms of domain information. While a lot can be expressed using UML diagrams, it is often not possible to express every detail using UML diagram elements alone.

2.2.5 The OCL and the UML

Taking an existing UML diagram and adding OCL constraints leaves the existing diagram intact, but adds information used to generate PSMs and source code. The OCL can be added to provide information to a UML diagram. The following is a simple list, of some situations where additional information is needed[45].

- Elements might be underspecified. For instance, when an attribute has been marked as derived, but no rule is given. The value of a derived attribute is produced by using one or more previously defined elements in the system. The rule indicates how the derived attribute's value is calculated.
- Business rules need to be added to the model. Business rules usually result in a number of invariants[7].
- A more precise definition for operations in a system (the system interface). These definitions can be specified by using preconditions and postconditions.
- Some aspects of diagrams may appear to be ambiguous. The ambiguity can be removed by the specification of invariants.

The OCL can be used to enrich UML diagrams. We will now look at each of the major UML diagrams to see how the OCL can be used.

Class Diagrams and the OCL

The most important diagram in a model is the class diagram. The class diagram is important because all models are structured around classes or components. The class diagram is where all structural information in the model is situated. The use of the OCL relies on types defined in the class diagram.

Derivation Rules

The class diagram may define attributes and associations, which are derived from other model elements. These derived elements cannot stand alone. The value of these derived elements must always be determined by other values in the diagram. Leaving out the derivation for a derived element creates an incomplete model. In the OCL, we can use a derivation rule to specify the missing derivation information. Figure 2.16 illustrates such a derivation rule. The rule states that the derived attribute “FullName” consists of a concatenation of the customer’s first name with the customer’s last name. Figure 2.16 also illustrates the use of the “concat” operation defined on the “String” type. The “concat” operation results in a new string that holds the value of the calling string concatenated with the string provided as the argument. For example, if the “firstName” attribute contained a value of “David” and the “lastName” attribute contained a value of “Arnold”, the value of the “FullName” attribute would be “David Arnold”.

```
context Customer::FullName : String
derive: firstName.concat(' ').concat(lastName)
```

Figure 2.16: Example derivation rule to complete an incomplete UML model

Syntactically, a derivation rule is defined by a context declaration followed by an OCL expression, which defines the rule. The context declaration consists of the path to the attribute followed by the attribute type. Following the context declaration, the *derive* label is used to indicate the start of the OCL expression which makes up the derivation. The expression must conform to the attribute type defined in the context declaration. The types will be checked by the compiler during the semantic pass. The details of which will be presented in Chapter 5.

Initial Values

The UML can be used to specify the initial value of an attribute. If the initial value is missing from a UML diagram the OCL could be used. Figure 2.17 illustrates an example of an initial value expression. The figure states that the initial value of the “purchases” attribute contained within the “Customer” class is zero. The difference between an initial value and a derivation rule is that the initial value only specifies the starting value of an attribute. The derivation rule acts more like an invariant, stating how to compute the value of the attribute each time it is used.

```
context Customer::purchases : Integer
init: 0
```

Figure 2.17: *Example initial value expression*

Syntactically, an initialization rule is defined by a context declaration followed by an OCL expression, which defines the rule. The context declaration has the same structure as the context declaration used for derivation rules. Following the context declaration, the *init* label indicates the start of the OCL expression, which yields the initial value. The initial value must conform to the attribute’s type. Initial and derivation rules may be mixed together after a single context declaration. Figure 2.18 illustrates such an example.

```
context Person::income : Integer
init: -- Initial value of a person’s income
derive: -- Expression to calculate a person’s income
```

Figure 2.18: *Example initial value and derivation rule expression*

Query Operations

The class diagram can introduce query operations. Query operations are operations that do not change the system’s state. That is, they have no side effects. Query operations result in a value, or a set of values. The OCL can be used to define query operations. The OCL expression is known as a body expression. Figure 2.19 illustrates a body expression. The body rule in Figure 2.19 states that the query called “GetCustomerName” is defined by a call to the “FullName” derived attribute.

Syntactically, a query operation, or body rule, is defined by a context declaration followed by the actual body expression. The context declaration consists of a classifier name followed by the operation name.

```
context Customer::GetCustomerName() : String
body: FullName
```

Figure 2.19: *Example body expression*

If the operation has parameters, then a parameter list is specified between two parentheses. If no parameters are required, then an empty parenthesis pair is given. Finally, the context declaration defines the return type for the operation. Following the context declaration, the *body* label is used to specify the start of the OCL expression. The result type of the OCL expression must match the return type defined in the context declaration.

Since OCL expressions can handle any value or collection of values defined in a system[45], the OCL has the same abilities as the SQL[1]. This can be shown as the body of an SQL expression can be expressed via a single OCL expression. From this information, we can clearly see that the OCL is both a constraint language and a query language[1, 45].

New Attributes and Operations

Most elements in a model are introduced in UML diagrams. Attributes and operations can also be added to the model using an OCL expression. An attribute that is defined via the OCL is always a derived attribute. Figure 2.20 illustrates such an attribute definition. The attribute defined in Figure 2.20, is named “salary” and is of type “Real”. The value of the attribute is defined by taking the salary attribute for each of a person’s jobs and summing it together, creating the total salary for a person. Details of the “sum” operation will be presented in Chapter 5.

```
context Person
def: salary : Real = jobs.salary->sum()
```

Figure 2.20: *Example attribute definition in the OCL*

Syntactically, the attribute is defined with a context declaration followed by the actual definition. The context declaration is the class to which the attribute is added. The *def* label indicates the start of the actual definition. The expression that defines the attribute includes the name and type of the attribute, followed by the expression, which defines the attribute.

Operations can be defined similarly to attributes. An operation that has been defined in an OCL expression is always a query operation. Figure 2.21 shows the definition of a new operation named “getJobs”. The figure introduces the operation “getJobs”, which returns a set of Job objects that are linked to the current person instance. Syntactically, operations are defined similarly to attributes. The difference is that the operation name ends in parenthesis. If the operation has parameters, they are specified in the parenthesis. Figure 2.21 makes reference to one of the built-in OCL collection types: Set. Details for the OCL collection types are provided in Chapter 5.

```
context Person
def: getJobs(): Set(Job) = self.jobs
```

Figure 2.21: Example operation definition in the OCL

Invariants

Another way to specify additional information on a class diagram is through an invariant. In the OCL, an invariant is specified using a boolean expression. If the expression evaluates to true, the invariant is met. Figure 2.22 shows an invariant. The invariant states that a customer must have an age of at least eighteen years. An invariant may also be named. Naming an invariant may be useful for documentation or for reference. The invariant in Figure 2.22 is named “ofAge”. More information on invariants, is provided in Section 2.1.1.

```
context Customer
inv ofAge: age >= 18
```

Figure 2.22: Example invariant

Preconditions and Postconditions

Adding preconditions and postconditions to operations is another way to add specification to a class diagram. As preconditions and postconditions specify a contract for an operation, they are an effective way to define the interface for a class. Section 2.1.1 showed how the OCL can be used to specify preconditions and postconditions.

Cycles in Class Models

Some class diagrams contain cycles. A cycle is the situation where you can start in one class instance, and by navigating through various associations, you can return to the instance that you started in. These cycles often create ambiguities[45]. Figure 2.23⁶ shows an example of a class diagram with a cycle. In the figure, a “Person” owns a “House”. A “House” is paid for by taking out a “Mortgage”. The “Mortgage” takes as security the “House”, which is owned by the “Person”. We will assume that the flaw in the diagram is that a person can have a mortgage, which takes as security a house that is owned by a different person. This is clearly not the intention of the diagram. The OCL can be used to write a simple invariant, which will fix the flaw. Figure 2.24 shows such an invariant. The invariant states that the mortgages held by a person, must use a security item (house) that is owned by the same person. Cycles in a class diagram should be checked, and any constraints on a cycle should be stated as an invariant on one of the classes in the cycle.

Figure 2.23: UML class diagram with a cycle

```

context Person
inv: self.mortgages.security.owner->forAll(owner:Person | owner = self)
  
```

Figure 2.24: Class invariant to resolve the cycle

⁶Figure taken from [45].

Dynamic Multiplicity

Associations in a class diagram can lead to imprecise specifications. These imprecise specifications occur when the multiplicity is based on another value in the system. This is called dynamic multiplicity. Consider Figure 2.25. The class diagram tells us that the movies can have any number of moviegoers. Of course this is not the case, as a given movie theater can only hold a pre-determined number of people.

Figure 2.25: UML class diagram that contains dynamic multiplicity

We can specify this restriction via an invariant. Figure 2.26 illustrates a valid invariant. The invariant specifies that the number of moviegoers for a particular movie must be less than or equal to the number of seats in the movie theater. Thus, the invariant directly specifies the multiplicity on the number of moviegoers.

```

context Movie
inv: moviegoers->size() <= theatre.numberOfSeats
  
```

Figure 2.26: Class invariant to resolve dynamic multiplicity

Optional Multiplicity

When a class diagram contains optional multiplicity, it is often a partial specification of what is really intended (Figure 2.23). In most cases, the optional association is bound to some other element in the system. In the case of Figure 2.23, if a person has a mortgage, then that person owns a house. This constraint can be specified by using the invariant in Figure 2.27. If optional multiplicity is used in the

class diagram, an invariant should be used to define precisely when the association should, and should not be empty.

```
context Person
inv: mortgages->notEmpty() implies houses->notEmpty()
```

Figure 2.27: *Class invariant to resolve optional multiplicity*

Sequence Diagrams and the OCL

OCL expressions can be used to complete sequence diagrams. In a sequence diagram, instances are shown. In the UML, we have a requirement that our model must be internally consistent[33, 34]. Each instance shown must be of a type that is defined in another diagram, such as the class diagram. With the definition of these types, the OCL can be used to provide additional information to our sequence diagrams. For example, a signal in a sequence diagram may only be sent if a given condition is satisfied. The OCL can be used to specify this condition. The OCL expression is then propagated, with no transformation, to a state machine diagram and is used as a guard.

Statecharts and the OCL

In UML statecharts, OCL expressions can be used in several ways. The following sections present a brief overview of how they can be used.

Guards

A guard is a condition on a transition in a statechart. When a signal is raised, it fires a transition, and the guard is checked to see if the transition should be executed. The OCL can be used to specify guard expressions for transitions.

Change Events

Change events are events that are generated when one or more attributes change value. An event occurs when a condition specifying the value of an attribute changes from false to true. A change event condi-

tion can be specified by an OCL expression. For example, an OCL expression could be used to test an attribute in the system. The result of evaluating the OCL expression would be a boolean value. If the result was true a change event would be fired. A change event is linked directly to a transition. When the change event fires, the transition is executed.

It should be noted that a change event is different from a guard condition. A guard is checked when an event linked to the guard's transition fires. If the guard is false, the transition is not executed, and the event is finished. When a change event occurs, a guard can still block the execution of the transition linked to the change event.

Restrictions on States

In most cases, a certain state in a statechart should only be achieved when specific conditions evaluate to true. This can be enforced by placing invariants on states. The OCL expression in Figure 2.28 illustrates this. The expression states that if the "Bottle" class is in the "empty" state, then its contents must be equal to zero. A state of a given object can be tested by using the "oclInState" operation[32]. Implementation of the "oclInState" operation is presented in Chapter 5.

```
context Bottle
inv: self.oclInState(empty) implies contents = 0
```

Figure 2.28: Invariant placed on a state

Activity Diagrams and the OCL

The OCL can be used in activity diagrams in the same way as is used in sequence diagrams[45]. As activity diagrams are used to specify the flow of a complete system rather than a specific object, the OCL keyword *self* cannot be used.

Component Diagrams and the OCL

In a component diagram, there is not much use for OCL expressions. OCL expressions are used only when elements usually defined in a class diagram are present in a component diagram. In this case,

the OCL is used in the context of preconditions and postconditions on component interfaces. These preconditions and postconditions are the same as those in the class diagram used to define the classes involved in the component interface.

Use Cases and the OCL

In use cases, preconditions and postconditions can be used. These preconditions and postconditions can be expressed using the OCL. Use cases provide an informal way to express requirements. As the OCL is a formal language, we need to make some adjustments so that the OCL can be used in the context of use cases[45].

Use cases can be considered operations defined on a complete system[45]. However, the complete system cannot be identified as a single type. This means, that preconditions and postconditions cannot have any contextual type of contextual instance. The consequence is that the keyword *self* cannot be used[45].

2.2.6 Summary of the OCL, the UML, and MDA

The OCL provides a mechanism for adding information to the various UML diagrams. The information added removes ambiguities from the diagrams, simplifies the diagrams, and adds information that cannot be expressed within the context of the diagram. The additional information added by the OCL is vital to MDA. This additional information at the PIM level aids in automating the transformation to the PSM level. Without the information provided by the OCL, the transformation to a PSM cannot be automated. Without the OCL, additional information would have to be inserted to complete the transformation process. The OCL can then be taken from the PSM level and translated to provide additional information to the code generator.

2.3 Introduction to .NET

In early 1998 Microsoft⁷ began the .NET⁸ initiative[23]. Unfortunately, there seems to be a lot of confusion in many circles with respect to what .NET actually is and what it does. The following sections will present an overview of the technology, the common language runtime (CLR), the framework class library (FCL), and the shared source common language infrastructure (CLI)[26, 38, 40].

2.3.1 What is .NET?

The .NET framework consists of two major components: the common language runtime (CLR)[23, 26, 38] and the .NET framework class library (FCL)[23, 26, 38, 46].

The CLR abstracts the operating system's services, and serves as an execution engine for managed applications. A managed application is a program written using a .NET compatible language. Every action of this language is subject to approval by the CLR before executing. This process is done at runtime, on a systematic basis as the managed application is executing. These properties make the CLR sound much like an interpreter. The CLR is not an interpreter. .NET compatible languages are not interpreted, they are compiled. The CLR will be examined in more detail below.

The FCL is an object-oriented API that managed applications write to. The FCL provides over seven thousand classes. These classes encapsulate existing technologies, such as the Windows API, MFC, ATL, and COM. Anything that can be done with these technologies is now done via the FCL.

A Windows API function or a COM object can be called from within a managed application. Performing such an operation requires transitioning from code run in the CLR (managed code), to native machine code that runs without the runtime's help (unmanaged code). Such a transition comes with two drawbacks. The first is that depending on the platform, the native machine code you want to call may not exist. For example, a Windows API function does not exist on machines without Windows. The

⁷ActiveX, Jscript, Microsoft, MSDN, .NET, Visual Basic, Visual C++, Visual C#, Visual Studio, Windows, and Windows NT are registered trademarks of Microsoft corporation.

⁸Pronounced dot NET.

second drawback is that the transition from managed code to unmanaged code suffers with respect to performance.

2.3.2 The Common Language Runtime

The heart and soul of .NET is the common language runtime (CLR)[23, 26, 38]. If you were to draw a .NET model, it would look like an upside down pyramid. At the top would be the development environments, like Visual Studio .NET. Next, would come the high-level .NET compatible languages like Visual Basic .NET, Visual C++ .NET, and Microsoft's new .NET specific language: C#. Below the high-level languages we find the intermediate language (IL)[23], and the framework class library (FCL). Last, but not least, at the tip of the pyramid, we find the CLR. The CLR sits on top of the operating system. Every byte of code that is written, is either executed by the CLR or is given permission by the CLR to run outside, as in the case of unmanaged code. Instead of the CLR understanding all of the .NET compatible languages, it only understands one: the intermediate language (IL). When any high-level language, such as C#, is compiled, it is translated into a pseudo assembly language. The assembly language is the IL. This IL is the result of a successful compile. The IL can be seen as more of a pseudo assembly language because it is not geared towards any specific processor. The processor is the CLR. When the application runs, the IL is loaded by the CLR and executed.

All .NET applications start by writing source code in any .NET compatible language. Figure 2.29⁹ illustrates the compilation and execution process. Microsoft provides .NET compilers for five languages: C#, J#, C++, Visual Basic, and JScript. Microsoft has also provided a software development kit for making your own .NET compiler. There are currently .NET compilers for a wide range of languages, from Cobol to Eiffel[42] to Perl.

The output of a .NET compiler is a managed module. A managed module is an executable or library, which contains two parts: IL code and metadata. As stated before, the IL code is a pseudo assembly representation of corresponding source code.

⁹Figure taken from [23].

Figure 2.29: The creation and execution of a managed .NET application

Metadata can be seen as a directory of all the types, methods, and properties implemented in a module's IL¹⁰. Metadata allows the CLR and other modules to determine what exactly a given managed module has to offer. It also determines any dependencies that the managed module requires in order to execute. Managed modules can be self sufficient, or can depend on any number of other managed modules. These dependencies are not required to reside on the same machine or platform. Along with the required metadata, developers can include metadata of their own, in the form of attributes that can be read by any other managed module, all at runtime. As an example for the use of metadata, a C# compiler can look inside a library containing a class written in another .NET compatible language and use it as a superclass for a new C# subclass.

Once created, a managed module is executed via the CLR. Unlike Java¹¹, we do not have to type something like "java mymodule.exe" to make it run. A managed module is executed as if it were a

¹⁰Refers to interfaces, classes, structures, enumerations, and delegates (wrappers around callbacks).

¹¹Java is a registered trademark of Sun Microsystems.

native image. An operating system-specific stub contained within the module's header is executed. The operating system-specific header starts the CLR, which invokes the loader. The loader loads the managed module and any required dependencies into a new application domain. An application domain is a virtual memory space where the managed application is allowed to operate. The CLR will strictly enforce the boundaries of the application domain, and will throw a runtime exception (similar to Java), if a managed application attempts to violate this space. Unlike Windows NT/2000/XP, the CLR does not host an application in its own process. The CLR may place several application domains in the same process. There are no security or safety issues here because every application domain is running managed code. This allows the CLR to keep everything separate. The advantage of not having each application domain in a different process is that processes are heavy weight in terms of system resources. The reduced process execution model is especially beneficial for web servers. The web servers are no longer required to spawn a separate process for each incoming request. The CLR instead creates few processes, each hosting several application domains.

Once a managed module and its dependencies are loaded, the CLR will perform type and security checking on the loaded module, and report any errors before executing. The validation process is complex and not within the scope of this work[23, 40]. Upon successful completion of the loading and validation processes, the IL code is just-in-time (JIT) compiled. The JIT compilation operation is where the CLR differs from an interpreter. The JIT compiler actually compiles the IL code into native machine code. The CLR executes the native machine code. The JIT compiler works on demand. That is, methods that are never called are never compiled. The JIT compiler uses a cache so that a method is JIT compiled only once, regardless of how many times the method is called. The only drawback is that if the JIT cache is small, as is the case in PDAs, the same method may need to be JIT compiled more than once. To reduce some of the performance drawbacks associated with the JIT step, the JIT compiler optimizes the IL code based on the target platform with respect to the underlying hardware and operating system. To further reduce performance drawbacks you can use the NGEN utility¹² to perform the JIT step once and create an image file¹³ in native code for the target platform. Some software installation programs¹⁴

¹²NGEN is a native code generator which is part of the .NET Framework SDK.

¹³Executable image file (exe).

¹⁴Microsoft's Installer (MSI) can perform this function.

will take the IL code and use the NGEN utility at install-time to create a platform specific image on the target machine. Doing so completely removes the JIT step from the execution process.

One last, but very important, feature of the CLR is that it maintains a garbage collector. All managed code is garbage collected. The garbage collector is maintained by the CLR. The details of the garbage collector are well out of scope for this thesis[23, 40]. Leaving the realm of the CLR, our focus will turn to the second major component of the .NET framework: the framework class library (FCL).

2.3.3 Framework Class Library

When developing software, predefined libraries or classes, such as the STL, or MFC, are used to make the development process cheaper and more time efficient. In .NET, everything is placed inside a single library: the framework class library (FCL)[26, 38, 46]. The FCL is an extensive class library. In order to organize the large number of classes, Microsoft has divided the FCL into about one hundred different namespaces. Table 2.1¹⁵ shows the most common namespaces.

Namespace	Contents
System	Core data types and auxiliary classes
System.Collections	Hash tables, resizable arrays, and other containers
System.Data	ADO.NET data access classes
System.Drawing	Classes for generating graphical output (GDI+)
System.IO	Classes for performing file and stream I/O
System.Net	Classes that wrap network protocols such as HTTP
System.Reflection	Classes for reading and writing metadata
System.Runtime.Remoting	Classes for writing distributed applications
System.ServiceProcess	Classes for writing Windows services
System.Threading	Classes for creating and managing threads
System.Web	HTTP support classes
System.Web.Services	Classes for writing web services
System.Web.Services.Protocols	Classes for writing web service clients
System.Web.UI	Core classes used by ASP.NET
System.Web.UI.Controls	ASP.NET server controls
System.Windows.Forms	Classes for GUI applications
System.Xml	Classes for reading and writing XML data

Table 2.1: Commonly used FCL namespaces

¹⁵Table taken from [46].

As you can see from Table 2.1, Microsoft has packed several rich classes into the FCL. With the FCL containing so many classes, some with hundreds of methods, properties and attributes, there is much to learn. While the STL is useful when writing C++ programs, it does not offer much help with writing a Visual Basic program. The FCL is available with all .NET compatible languages. Once you learn the FCL for C#, that knowledge can be used in Visual Basic .NET, or even Cobol .NET. The FCL requires a slight investment of time to get started, especially when familiar with other libraries. The rewards that the FCL brings will make the initial investment a good one. Our implementation makes use of the framework class library for mapping the OCL constructs to C#, as well as to implement the actual compiler. As we will be providing code level mappings, the next section will provide a concrete example of the IL syntax.

2.3.4 Hands On .NET

Squaring a Number in IL

We will now look at a piece of code written in intermediate language (IL). All .NET applications could be written in IL, however doing so has all the disadvantages of writing a Windows application in assembler. As we will be looking at modifying a C# compiler and the code generator, we will look at a low-level example. Figure 2.30 shows the complete code listing. The example shown is a simple managed application written directly in IL. The example program shown in Figure 2.30 consists of asking a user to enter a number, and then returning the number squared. The most common IL instructions are shown in Table 2.2.

Let us look at the code in more detail. The first line acts like a C++ “#include” statement. This tells the CLR that we will be referencing an external assembly called “mscorlib”. “mscorlib” is the core .NET runtime assembly. Virtually every .NET program will have a reference to “mscorlib”. Lines two and three define our assembly and module name. A module can contain one or more assemblies, but an assembly must be contained in a module. Line four defines our application class. Not everything in .NET has to be encapsulated in classes. Some .NET languages enforce such an encapsulation. Lines six, seven, and eight define and mark the start of the program’s entry point. Line nine defines the size

```

1) .assembly extern mscorlib { }
2) .assembly Ex1 { }
3) .module Ex1.exe
4) .class private auto ansi MyApp extends [mscorlib]System.Object
5) {
6)     .method private hidebysig static void Main() cil managed
7)     {
8)         .entrypoint
9)         .maxstack 4
10)        .locals init (int32 V_0)
11)        ldstr "Hello, please enter the number to be squared:"
12)        call void [mscorlib]System.Console::WriteLine(string)
13)        .try
14)        {
15)            call string [mscorlib]System.Console::ReadLine()
16)            call int32 [mscorlib]System.Convert::ToInt32(string)
17)            stloc.0
18)            leave.s end_catch
19)        }
20)        catch [mscorlib]System.FormatException
21)        {
22)            pop
23)            ldstr "Your input was not an integer."
24)            call void [mscorlib]System.Console::WriteLine(string)
25)            ret
26)        }
27)        end_catch:
28)        ldstr "{0} squared is {1}"
29)        ldloc.0
30)        box [mscorlib]System.Int32
31)        ldloc.0
32)        ldloc.0
33)        mul
34)        box [mscorlib]System.Int32
35)        call void [mscorlib]System.Console::WriteLine
            (string, object, object)
36)        ret
37)    }
38) }

```

Figure 2.30: *Squaring a number in IL (with line numbers)*

of the stack that the program needs to operate. The stack size will be used by the CLR to create the initial environment. Line ten declares the one and only local variable. This variable is a 32-bit integer. .NET supports 8, 16, 32, and 64 bit integers[46]. Lines eleven and twelve display the initial message to the user via, a call to the “WriteLine” function located in the “mscorlib” assembly that was imported in line one. Lines thirteen to nineteen create a “try” block. This reads a line of input via the “ReadLine” function, and then converts it to a 32-bit integer that is stored in the local variable from line ten. If the value entered cannot be converted to a 32-bit integer in line sixteen, a “FormatException” will be thrown. Lines twenty through twenty-six display an error message to the user and then terminate the

Instruction	Description
BOX	Converts a value type into a reference type
CALL	Calls a method; if the method is virtual, virtualness is ignored
CALLVIRT	Calls a method; if the method is virtual, virtualness is honoured
CASTCLASS	Casts an object to another type
LDC	Loads a numeric constant onto the stack
LDARG[A]	Loads an argument or argument address [A] onto the stack
LDELEM	Loads an array element onto the stack
LDLOC[A]	Loads a local variable or local variable address [A] onto the stack
LDSTR	Loads a string literal onto the stack
NEWARR	Creates a new array
NEWOBJ	Creates a new object
RET	Returns from a method call
STARG	Copies a value from the stack to an argument
STELEM	Copies a value from the stack to an array element
STLOC	Transfers a value from the stack to a local variable
THROW	Throws an exception
UNBOX	Converts a reference type into a value type

Table 2.2: *Common IL instructions*

program. Lines twenty-eight through thirty-five perform the actual squaring. By leaving the values on the stack, the result can be displayed to the user.

Squaring a Number in C#

As shown earlier, writing code in IL is only slightly simpler than writing a Windows application in assembler. Most .NET developers use a .NET compatible high-level language, such as C#, to write programs. Figure 2.31 shows the program in Figure 2.30 written in C#. As shown by the two figures, it is much easier to write code using a high-level language rather than using the IL directly.

This concludes our brief look into .NET code. Chapters 5 and 6 will illustrate how we have mapped each OCL expression to the .NET platform. The next section will look at the C# language in detail. More information on .NET can be found in [11, 23, 26, 38, 40, 46].

```
using System;
class MyApp
{
    static void Main()
    {
        int input;
        Console.WriteLine("Hello, please enter the number to be
            squared:");
        try
        {
            input = Convert.ToInt32(Console.ReadLine());
        }
        catch(FormatException)
        {
            Console.WriteLine("Your input was not an integer.");
            return;
        }
        Console.WriteLine("{0} squared is {1}", input, input*input);
    }
}
```

Figure 2.31: *Squaring a number in C#*

2.3.5 Shared Source Common Language Infrastructure

In the summer of 2001, Microsoft announced plans for a shared source version of the common language infrastructure (SSCLI)[39, 40]. The SSCLI was released in November 2002. The SSCLI contains a fully-functional .NET execution engine, a C# compiler, programming libraries, and other programming tools. Microsoft's rationale for releasing the SSCLI consists of three goals. The first is to validate the portability of the CLI standard. The second is to allow people to learn about and understand the commercial version of the CLR. The third, and most relevant goal is for the purposes of academia, is to facilitate long-term academic interest in the CLI, and the corresponding ECMA standards[12, 13, 14, 15, 16, 40].

The SSCLI package is not the same as Microsoft's .NET Framework commercial release. The SSCLI does not contain code related to Windows development, web applications, and data access. The SSCLI only contains C# and JScript compilers. The SSCLI contains a platform adaptation layer for portability. The platform adaptation layer is not contained within the commercial release. There are also several development and debugging tools which are contained exclusively with the SSCLI package[39, 40].

The SSCLI is not the only shared source version of the .NET runtime. There are also two third-party shared source implementations. These implementations are based on the ECMA standards[12, 13, 14,

15, 16]. One implementation is the Mono project from Ximian[47], which will be discussed in the next section. The second implementation is the DotGNU project[6]. The DotGNU project was not considered because it is not compatible with Microsoft's .NET runtime. If the DotGNU project was used, the resulting executables would only run using the DotGNU runtime. For the purposes of this research, only the Mono project will be used for discussion and development.

2.3.6 The Mono Project

The Mono project is an open source effort sponsored by Novell to create a free implementation of the .NET Framework[47]. The project is maintained by Ximian. The Mono project includes a C# compiler, a common language runtime, and a set of class libraries. As the Mono C# compiler can compile C# code to run on Microsoft's runtime, only the C# compiler portion of the Mono project will be considered.

Mono's C# compiler is considered feature complete, with respect to version 1.0 of the C# specification[11]. The Mono C# compiler is written entirely in C#. The C# compiler found in the SSCLI is written in C++. There are two source level advantages to using the Mono compiler over the SSCLI compiler. The first is that C# does not use pointers. Not having to deal with pointers and memory allocation/de-allocation makes development and debugging of our compiler easier. The second advantage is that the SSCLI compiler is procedural in nature, while the Mono compiler is object-oriented and well structured.

It should be noted that the Mono compiler is in its infancy, and that while feature complete, it does not have the same level of optimization or performance as the SSCLI compiler. For the purposes of this research, optimization and performance are not major considerations. Thus, as stated before the Mono C# compiler will be used as the base for our C#/OCL compiler.

2.3.7 .NET Summary

The .NET initiative is a large one, and we have just scratched the surface with respect to what .NET has to offer. .NET is much more than just a new technology; it is a new way of thinking. Things that most development cycles focus on, such as distribution and web enabling, are a non-issue with the FCL. With

all .NET languages compiled into IL, a programmer can use any language (or languages) that they feel comfortable using. .NET allows developers to focus on what their applications do, rather than worrying about the how. We will now take a detailed look at a language designed explicitly for .NET: C#.

2.4 C#

To keep in line with the OCL's ease of readability, a complementary high-level language to integrate with the OCL was chosen. In short, a language that is easy to write, read, and maintain like Visual Basic, but still provides the power and flexibility of C++. This section will illustrate that Microsoft's new .NET specific language C# fits this description perfectly.

C# was standardized by the ECMA in December 2002[11, 46]. C# is a robust, high-level language with built in type-safety, garbage collection, simplified type declarations, versioning and scalability support. The following sections provide detailed information regarding the C# language. The sections are provided for readers who are not familiar with C#. Readers with a background in C# may wish to skip these sections. Details pertaining to the integration with the OCL are not introduced until Chapter 4.

2.4.1 C# Overview

The main design goal of C# is simplicity rather than pure power[43]. In order to achieve this simplicity there is a decrease in runtime performance, compared with languages designed for speed (C/C++). In return for the loss in performance, C# provides type-safety and automatic garbage collection. One of the major differences between C# and C++ is that C# does not distinguish between a value type and a reference type.

In C++, the “.” operator is used for navigating a value type and the “->” operator is used for reference types. In C#, all navigations are performed via the “.” operator. This one-to-one operator navigation allows for a direct mapping between the two languages. Meaning, a transformation from a group of C#

operators to a single OCL operator is not required. The reader should note that C# supports both value and reference types, but only uses a single operator (“.”) for member access.

In C++, there is a notion of signed, unsigned, and other type modifiers; C# only uses the type name. It is the compiler’s responsibility to determine the number of bits needed to store a value. This notion of simple types makes it easier to derive mappings between the OCL’s four primitive types to C#’s fifteen.

Like Java, C# provides a base type: *object*. The base type allows each type in the language to be viewed as an object. The *object* type is analogous to the *OclAny* type found in the OCL. Like C++ and Java, classes in C# are grouped into namespaces. The notion of an OCL package can be mapped into C# namespaces.

C# is a modular and object-oriented programming language. Unlike C++, C# supports true interfaces, and does not allow for global methods, global attributes, and global constants. In addition, C# does not support multiple inheritance. It does, however, support the realization of multiple interfaces, similar to Java.

The C# language includes several additional concepts as first class entities. These include properties, events, structures, and delegates. We will now look at these entities, along with others in the following sections.

2.4.2 Types

As stated before, C# supports both value types and reference types. Value types include simple types, such as integers. Enumerations and structures are also considered value types. Reference types include classes, interfaces, delegates and array types[11]. The difference between a reference type and a value type is that a value type directly contains its data. A reference type contains a reference to an object. It is possible for more than one reference type to reference the same object. That is, an operation on a reference type may affect other reference types. Value types each store a copy of the data. Thus, it is not possible for operations on one value type to affect another value type. Figure 2.32 illustrates the difference between reference and value types. The assignment to *val1* does not affect *val2* because

both *val1* and *val2* are value types. On the contrary, the assignment to *ref2* affects *ref1* as both variables reference the same object.

```
using System;
class Class1
{
    public int Value = 0;
}
class Test
{
    static void Main()
    {
        int val1 = 0;
        int val2 = val1;
        val2 = 123;
        Class1 ref1 = new Class1();
        Class1 ref2 = ref1;
        ref2.Value = 123;
        Console.WriteLine("Values: {0}, {1}", val1, val2);
        Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
        // Output:
        // 0, 123
        // 123, 123
    }
}
```

Figure 2.32: *Differentiating between value and reference types*

As stated previously, C# provides a unified type system. That is, all of the types in C# are derived from the *object* type. It is possible to call any method defined on the *object* type, on all types, both reference and value. Even primitive types can be used to invoke methods defined on the *object* type. Figure 2.33 illustrates such a case.

If a value type needs to be used as a reference type, an object box is created to hold the value, and then the value is copied into the box. This operation is known as boxing. Unboxing is the opposite; the value is copied out of the object box. C#'s type system along with boxing and unboxing provide all of the benefits of objects without the extra overhead. For example, in applications where integer values do not need to be used as objects, they are represented by a 32-bit value. In applications that need integers to be objects, the conversion can be made on demand.

```
using System;
class Test
{
    static void Main()
    {
        Console.WriteLine(3.ToString());
    }
}
```

Figure 2.33: *Use of the object type*

2.4.3 Classes

A class is a data structure that may contain data members, function members, and nested types[11]. A data member may be either a constant or a field. Function members include methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors. Classes support inheritance, a mechanism that allows a derived class to extend and specialize a base class[11]. An instance of a class is always a reference type. The following sections will examine the specialized class types.

Abstract Classes

Unlike C++, C# allows for pure abstract classes. An abstract class is specified using the *abstract* modifier. Abstract classes can specify abstract members. These are members that derived classes must implement. Methods defined within an abstract class, which are implemented by derived classes, must be marked abstract and have no implementation in this abstract class. As with Java, it is not legal to instantiate an abstract class. Figure 2.34 shows an example of an abstract class, “Shape”. The “Shape” class contains an abstract method, “Draw”. The subclass of “Shape”, “Square” must implement the “Draw” method.

Sealed Classes

Contrary to abstract classes that require derivation in order to be used, a sealed class cannot have any subclasses. That is, a sealed class precludes derivation. A class can be marked sealed by using the *sealed* modifier as shown in Figure 2.35.

```
abstract public class Shape
{
    abstract public void Draw();
}
public class Square : Shape
{
    public void Draw()
    {
        // Draw the shape
    }
}
```

Figure 2.34: *An abstract class*

```
sealed public class MyClass
{
    ...
}
```

Figure 2.35: *A sealed class*

2.4.4 Properties

C# provides support for the enforcement of encapsulation: properties. A property is a member that provides access to a characteristic of an object or class[11]. Properties function in a similar manner to get/set methods in Java. Properties are similar to fields[46]. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have accessors that specify statements to be executed when their values are read or written. Figure 2.36 provides an example of a “Button” class, which defines a “Caption” property.

```
public class Button
{
    private string caption;
    public string Caption
    {
        get { return caption; }
        set
        {
            caption = value;
            Repaint();
        }
    }
}
```

Figure 2.36: *A Caption property defined on the Button class*

A property is specified by two separate code blocks. The “get” block specifies code for accessing the value of the property. In Figure 2.36, the value of the “caption” field is returned, but any group of C#

statements could be used. The “set” block specifies code for setting the value of the property. The actual value to be set is stored in the *value* keyword. The value can then be checked and/or manipulated during the specification process. It is possible to specify properties that are read-only or write-only. A read-only property only contains a get accessor. A write-only property only contains a set accessor. A property is used as if it were a field. Figure 2.37 illustrates the use of the “Caption” property defined in Figure 2.36.

```
Button b = new Button();  
b.Caption = "ABC";    // set; causes repaint  
string s = b.Caption; // get  
b.Caption += "DEF";   // get & set; causes repaint
```

Figure 2.37: *Using the Caption property*

2.4.5 Delegates

A frequent occurrence in programming is the need to execute an action for which the exact method is unknown at design time. In C++, this issue is handled using function pointers. A function pointer represents a variable, which points to a method, which can be dynamically executed. In C#, the notion of a delegate was introduced to enable the dynamic specification of a method.

A delegate is a reference type to encapsulate a method with a specific signature and return type. An attribute whose type is defined by a delegate can be assigned any method that has a matching signature (parameter set and return type). The attribute can then be used to invoke the assigned method. There are three steps for using delegates. The first step is to declare the delegate. A delegate declaration is a type declaration that declares a new delegate type. An example of a delegate declaration can be seen in Figure 2.38. The type declaration defines a delegate which accepts methods that return an integer value, and accept two parameters of type integer and double respectively.

```
public delegate int MyDelegate(int a, double b);
```

Figure 2.38: *Declaration of a new delegate type*

The second step in the use of delegates is the instantiation of the delegate type defined in the first step. As a delegate is a type in the system, it is instanced as a reference type. A delegate constructor takes the method that the delegate will represent as a parameter. Figure 2.39 illustrates the instantiation of

the new delegate: “MyDelegate”. The first instantiation represents a static method, while the second instantiation represents an instance method. The delegate “d3” represents the contents of both “d1” and “d2”. When “d3” is invoked, the contents of “d1” will be invoked, followed by the contents of “d2”.

```
public class DelegateTest
{
    public static int Method1(int a, double b)
    {
        ...
    }
    public int Method2(int a, double b)
    {
        ...
    }
    public static void Main()
    {
        MyDelegate d1 = new MyDelegate(Method1);
        DelegateTest dt = new DelegateTest();
        MyDelegate d2 = new MyDelegate(dt.Method2);
        MyDelegate d3 = d1 + d2;
    }
}
```

Figure 2.39: *Instantiation of a delegate type*

The third and final step to use a delegate is invocation. C# provides special syntax for invoking a delegate. When a delegate instance only contains a single method, the method is invoked with the arguments given to the delegate, and the method result is returned to the delegate. When a delegate instance contains more than one method, each method is invoked synchronously, with the same arguments. The result of the delegate invocation is the result from the last method. If a method throws an exception that is not handled, any methods following the throwing method are not invoked. Figure 2.40 illustrates the invocation of the three delegates shown in Figure 2.39.

```
int result = d1(1, 2.2); // Invokes Method1(1, 2.2)
d2(2, 4.4);             // Invokes dt.Method2(2, 4.4)
result = d3(4, 8.8);    // Invokes Method1(4, 8.8) and then
                        // dt.Method2(4, 8.8).
                        // Returns the result of dt.Method2 only.
```

Figure 2.40: *Invocation of delegate instances*

Technically, a delegate declaration defines a class that is derived from the “System.Delegate” class. A delegate instance encapsulates one or more methods, each of which is referred to as a callable entity[11]. For instance methods, a callable entity consists of an instance and a method on that instance. For a static method, a callable entity only consists of a method. Given a delegate instance and an appropriate set

of arguments, all of the methods assigned to the delegate can be invoked in a single statement. An important feature of delegates is that an instance of a delegate has no information about the classes of the methods that the delegate encapsulates. The only important thing is that the methods encapsulated by the delegate match the delegate's type. This feature makes delegates a perfect match for anonymous invocation.

2.4.6 Events

C# recognized the popularity of event-driven programming and made the event concept a first class entity. An event notifies interested parties that something happened. It is the notified object's responsibility to act in accordance. For example, in a graphical user interface (GUI), clicking a button will raise an event notifying the interested parties that the button was clicked.

In C#, events and delegates are tightly coupled concepts. In order for the event handling to be as flexible as possible, the event notification must be dispatched to the proper event handler(s). The event handler is usually defined as a delegate. The C# implementation of the event concept follows the Observer pattern[10], where an object publishes a set of events to which other objects subscribe (observers).

A class defines an event by providing an event declaration (an attribute definition, with an added *event* keyword), and an optional set of event accessors. The type of an event declaration must be a delegate. An event maintains a list of event handlers (delegates), all of whom are notified when the event occurs. Figure 2.41 illustrates the use of an event. The figure starts by defining a delegate that will be used for handling the events. Next, a "Button" class is defined to contain a click event. The click event stores delegates of type "EventHandler". The "LoginDialog" class contains two buttons. The constructor instantiates the buttons and assigns each one a handler. The handler will be called to handle the clicking of the respective button.

Outside of the owning event's class, an event can only be used on the left-hand side of the "+=" and the "-=" operators. As shown in Figure 2.41, the "+=" operator is used to add a new delegate to the event's delegate list. The "-=" operator provides the inverse function by removing a delegate from the event's delegate list. An event may be only invoked within its containing class. In the case of Figure 2.41, the

```

public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
    public event EventHandler Click;
}
public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;
    public LoginDialog()
    {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }
    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }
    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}

```

Figure 2.41: *Illustration of a button click event*

“Click” event can only be invoked from within the “Button” class. Figure 2.42 illustrates an appropriate method to fire the “Click” event.

```

protected void OnClick(EventArgs e)
{
    if(Click != null)
        Click(this, e);
}

```

Figure 2.42: *Firing a button click event (within the Button class)*

The action of firing an event is the same as calling a delegate. The only thing of note is that if the event invocation list is empty, the event will have a value of null, and calling a null event results in an exception. It is always good practice to check the event before using it, as shown as Figure 2.42.

2.4.7 Indexers

An indexer is a member that enables an object to be indexed in the same way as an array[11]. As already stated, properties enable field like access. Indexers enable array like access.

Indexer declarations are similar to property declarations, with the exception of the indexer name, and the

indexing parameters. Indexers are nameless.¹⁶ Indexers also include indexing parameters. The indexing parameters are provided between square brackets, similar to array access. Figure 2.43 illustrates an indexer with a single indexing parameter. As indexers reference an instance of a class, a static indexer results in a compile-time error.

```
public class StringList
{
    private string[] strings;
    public string this[int index]
    {
        get { return strings[index]; }
        set { strings[index] = value; }
    }
}
public class Test
{
    public static void Main()
    {
        StringList sl = new StringList();
        sl[0] = One;
        s2[1] = sl[0];
        s3[2] = Three;
    }
}
```

Figure 2.43: *Indexer with a single indexing parameter*

2.4.8 Operators

The C# language allows for adding standard operators (e.g. “+”, “-”, etc.) to a user-defined type. An operator is a static method whose return value is the result of the operation applied on the given parameters. Adding an operator to a user-defined type is known as operator overloading, and is similar to operator overloading in C++. To overload an operator, the keyword *operator* is used, and is followed by the operator to be overloaded as shown in Figure 2.44.

```
public class Integer
{
    public static Integer operator+(Integer lhs, Integer rhs)
    {
        // Add the two integers and return the result
    }
}
```

Figure 2.44: *Overloading of the + operator*

¹⁶The name used in the declaration of an indexer is *this*, since it is the instance of a class that is being indexed.

It is also possible to define type conversion operators. A conversion operator converts one type to a second type. C# distinguishes between implicit and explicit conversions. Implicit type conversion is used when the conversion is guaranteed to succeed. If an implicit type conversion is defined on a given type, a cast is not required to invoke the type conversion. An explicit type conversion requires an explicit cast in order to invoke the conversion. Explicit type conversions should be used when some information may be lost during the conversion. Figure 2.45 illustrates definition and usage of both implicit and explicit type conversions[26].

```
public class RomanNumeral
{
    private int value;
    // Implicit conversion operator
    static public implicit operator RomanNumeral(int value)
    {
        return new RomanNumeral(value);
    }
    // Declare an explicit conversion from a RomanNumeral to an int:
    static public explicit operator int(RomanNumeral roman)
    {
        return roman.value;
    }
}
class Test
{
    static public void Main()
    {
        RomanNumeral numeral;
        // Calls the implicit conversion to convert
        // the int to a numeral
        numeral = 15;
        // Calls the explicit conversion from numeral to int.
        // Because it is an explicit conversion, a cast must be used:
        Console.WriteLine((int)numeral);
    }
}
```

Figure 2.45: *Implicit and explicit type conversion operators*

2.4.9 The *new* Modifier

A class member declaration is permitted to declare a member with the same name and signature as an inherited member. When this occurs, the derived class member is said to hide the base class member[11]. That is, the derived class member, with the *new* modifier, replaces the base class member.

The only exception to the signature consistency rule is when the *new* modifier is used on attributes. An

attribute in a derived class can be of a different type than the hidden attribute of the base class. For example, in Figure 2.46, a class that represents a real number defines an attribute of type “float” to represent the actual numerical value. If the “Real” class is subclassed to represent an integer number, the numerical value may be of type “int”, with the same name as the “float” attribute in the “Real” class. This in effect, hides the base class attribute with an attribute of a different type.

```
public class Real
{
    private float _value;
}
public class Integer : Real
{
    private new int _value;
}
```

Figure 2.46: *Use of the new modifier*

2.4.10 Interfaces

C# allows for the definition of formal interfaces. Interfaces may contain methods, properties, events and indexers. Interfaces do not provide implementation for the members that are defined. The interface is merely a specification of members that must be realized by the implementing entities. In other words, an interface specifies a contract. A class or structure that implements an interface must adhere to the contract provided by the interface. An interface may inherit from one or more base interfaces. A class or structure may realize multiple interfaces. However they may only inherit from a single base class or structure. Figure 2.47 displays an example interface.

```
public delegate void StringListEvent(IStringList sender);
public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

Figure 2.47: *Example interface*

An interface member is sometimes referred to by its fully qualified name[11]. The fully qualified name of an interface member consists of the name of the owning interface, followed by a dot, followed by the name of the member. The fully qualified name for the first member in Figure 2.47 would be

“IStringList.Add”. If the interface resides within a namespace, the fully qualified name of the member also includes the namespace name.

An explicit interface member implementation is similar to a normal implementation. The difference being that the fully qualified member name is used instead of just the member name. Explicit interface member implementations have different accessibility characteristics than other members. As explicit interface members are never accessible through their fully qualified name, they are in a sense private. However, since they can be accessed through an interface instance, they are in a sense public[11]. Figure 2.48 illustrates the use of an explicit interface member implementation[11].

```
interface ICloneable
{
    object Clone();
}
interface IComparable
{
    int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}
}
```

Figure 2.48: *Explicit interface member implementation*

Explicit interface member implementations serve two purposes. The first is that because an explicit interface member is not accessible through the implementing class or structure, they allow interface implementations to be excluded from the public view. That is, an interface implementation can be hidden if it is of no use to the consumer of the class or structure. The second purpose is that explicit interface member implementations can be used to disambiguate between interface members with the same signature. Without explicit interface implementations, it would be impossible for a class or structure to have more than one interface member with the same signature[11, 46].

2.4.11 Structures

Structures are similar to classes in that they represent data structures that contain data members and functional members. However, unlike classes, structures are value types and do not require heap

allocation[11]. As previously stated, a structure type directly contains the data of the structure, whereas a class type class contains a reference to the data, known as an object. For this reason, structures are particularly suited to small data structures that have value semantics[46].

Structures do not support inheritance, initializers, destructors, or instance constructors which do not explicitly initialize each member. Figure 2.49 provides a simple example of a structure.

```
public struct Point
{
    public int X;
    public int Y;
    // Instance constructor okay because all members were initialized
    public Point(int pX, int pY)
    {
        X = pX;
        Y = pY;
    }
}
```

Figure 2.49: *Example structure definition*

2.4.12 Parameters

C# supports four types of parameters. The default parameter type that is used in the constructor shown in Figure 2.49 is an “in” parameter. An “in” parameter is considered a value parameter. The parameter refers to its own variable, one that is distinct from the corresponding argument[11]. Modifications to the parameter do not affect the original argument. The value parameter is initialized by copying the value of the corresponding argument.

A reference parameter is used for by-reference parameter passing. A by-reference parameter acts as a alias to the caller-provided argument[11]. A reference parameter does not define a variable, but rather refers to the argument. Thus, modifications to a reference parameter affect the original argument. A reference parameter is defined by using the *ref* modifier before the parameter declaration, as shown in Figure 2.50.

As Figure 2.50 illustrates, the *ref* modifier is required not only on the method parameter list, but also in calls to the method. The use of the *ref* modifier at the call site draws special attention to the parameter.

```
static void Swap(ref int a, ref int b)
{
    int t = a;
    a = b;
    b = t;
}
static void Main()
{
    int x = 1;
    int y = 2;
    Swap(ref x, ref y);
}
```

Figure 2.50: *Use of reference parameters*

This allows developers reading the code to see that the value of the parameter may change within the method call.

The third parameter type is an output parameter. An output parameter is similar to a reference parameter. The difference is the initial value of the parameter is ignored. Output parameters are especially useful when a method needs to return more than one value. An output parameter is defined by using the *out* modifier. Figure 2.51 shows the use of an output parameter. The only note with using output parameters is that a compile-time error will be generated if there is an execution path where any of the output parameters are not assigned. Each output parameter must be assigned a value before the method returns.

```
static void Divide(int a, int b, out int result, out int remainder)
{
    result = a / b;
    remainder = a % b;
}
static void Main()
{
    int ans, r;
    Divide(6, 4, out ans, out r);
}
```

Figure 2.51: *Use of output parameters*

For value, reference, and output parameters, there is a one-to-one correspondence between the arguments and the parameters[11]. In the case where the number of parameters is not known at development time, a parameter array can be used. A parameter array allows for a many-to-one relationship between the arguments and the parameters. This allows a parameter list to be used to handle variable length argument lists.

A parameter array is denoted by the *params* modifier[11]. There can only be one parameter array for a

given method, and it must be located at the end of the method's parameter list. The type of the parameter array must be a single dimensional array. As Figure 2.52 illustrates, the caller can pass zero or more arguments to the parameter array.

```
static void F(params int[] args)
{
    Console.WriteLine("# of arguments: {0}", args.Length);
    for(int i = 0; i < args.Length; i++)
        Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
}
static void Main()
{
    F();
    F(1);
    F(1, 2);
    F(1, 2, 3);
}
```

Figure 2.52: *Use of a parameter array*

2.4.13 Attributes

Up to this point, all the C# elements that have been examined focus on code generation elements. Upon compilation the elements examined so far translate into IL code. As already discussed, the IL code is just half of what makes up an executable module under the .NET runtime. The second half is metadata. We will now examine a C# element that contributes to the metadata of an executable module, rather than the IL code portion. That element is the attribute. The next section will examine the usage of attributes, followed by attribute extraction information.

Using Attributes to add Metadata

Attributes allow programmers to invent new kinds of declarative information or metadata[11]. Attributes can be attached to various programming elements. Attribute information can be retrieved at runtime. When an attribute is added to a program element, no IL code is generated. Instead, metadata is generated and added to the executable module. Figure 2.53 illustrates the separation between code, and metadata.

Attributes are defined through the declaration of attribute classes, which may have positional and named parameters. A positional parameter is defined by its location in an attribute's parameter list. A named

Figure 2.53: Code and metadata separation

parameter is defined by its name. Attributes are attached to entities in a C# program using attribute specifications, and can be retrieved at runtime as attribute instances[11].

Syntactically, attributes are placed within a set of square brackets, immediately before the program element that they are being applied to. The syntax for an attribute is similar to the syntax of a constructor, with the attribute's parameters passed as arguments within parenthesis. Named parameters are always passed following positional parameters. If a positional parameter is found after a named parameter, a compile-time error is generated. Figure 2.54 illustrates some attribute examples.

```

// Simple attribute usage
[Obsolete()]
public void Hide()
{
}
// Attribute with two positional parameters
[Obsolete("Use the new BubbleSorter class", true)]
class BubbleSort
{
}
// Attribute with a named parameter
[WebService(Namespace="http://www.ewesimplex.com")]
class MyWebService
{
}
  
```

Figure 2.54: Attribute declaration examples

C# provides several predefined attributes for use with compilation control, connection pooling, transactions, and serialization[26]. The user is also able to create attributes. All attributes inherit from the

“System.Attribute” class. Figure 2.55 shows the definition of a new attribute. An important thing to note in Figure 2.55 is that an attribute was used to help define our attribute. The “AttributeUsage” attribute is used to specify where and how a custom attribute can be used. In Figure 2.55, the “PluggableAttribute” can only be used on classes, and only one attribute per class can be used. Now that attribute creation and usage has been discussed, we will now turn to extracting attribute information out of the metadata.

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public class PluggableAttribute : Attribute
{
    // Constructor (one positional parameter)
    public PluggableAttribute(string type)
    {
        _type = type;
    }
    // Type accessor (read only)
    public string Type
    {
        get { return _type; }
    }
    // Notes accessor (read/write so we can set it
    // via a named parameter)
    public string Notes
    {
        get { return _notes; }
        set { _notes = value; }
    }
    // Internal representations for our values
    protected string _type;
    protected string _notes;
}
```

Figure 2.55: *Attribute definition example*

Extracting Attributes from Metadata

As stated before, attributes do not add any IL code to an executable module. The process of extracting metadata out of an executable module is called reflection. In C#, reflection can be performed via classes found in the “System.Reflection” namespace. Figure 2.56 illustrates the extraction of information from the attribute defined in Figure 2.55. As reflection is a rather complex concept, it will not be discussed any further in this thesis[11, 26, 46].

```

public class PluggableCheck
{
    // Default constructor; takes the type we are looking for our
    // Pluggable attribute in
    public PluggableCheck(Type theType)
    {
        _type = theType;
    }
    public string GetTypeName()
    {
        foreach(Attribute attrib in _type.GetCustomAttributes(true))
        {
            if(attrib is PluggableAttribute)
            {
                PluggableAttribute plug = attrib as PluggableAttribute;
                return plug.Type;
            }
        }
        return null;
    }
    // Internal storage for the type we are checking.
    protected Type _type;
}

```

Figure 2.56: *Extraction of attribute information from metadata*

2.4.14 C# and .NET

C# is one of several .NET compatible languages. Each .NET compatible language uses the same data types and utility classes. While this thesis contains information with respect to the integration of the OCL and C#, any .NET compatible language could be substituted with little modification.

2.4.15 Summary

The integration of the OCL into C# will allow for C# to retain all of its existing benefits, and gain all of the benefits provided by design by contract. The next section will look at some of the existing contract tools and technologies that currently exist.

2.5 Existing Tools and Technologies

In the context of design by contract and the OCL, a few tools and technologies are directly related to this work. The following section will briefly examine some of the more prevalent tools and technologies.

2.5.1 The Octopus Tool

Klasse Objecten has implemented one of two tools that fully supports version 2.0 of the OCL. The “OCL Tool for Precise UML Specifications” or Octopus, is a tool which is able to check the syntax of OCL expressions at the model level[19]. The tool also checks types, and for the correct use of model elements, like association roles and attributes.

The Octopus tool fully conforms to OCL version 2.0[19]. All the new constructs like initialization and derivation rules are supported. The Octopus tool also provides the ability to view OCL expressions in a SQL like syntax. This is particularly useful for those with a database background.

The Octopus tool was released on June 1st 2004. It is written in Java, and acts as a plug-in to the Eclipse framework[41]. Eclipse is required in order to run Octopus. Currently¹⁷, Octopus only runs at the model level, and does not support code generation. In the context of this thesis, the Octopus tool could be used at the model level. The model could generate code with the checked OCL expressions intact, and then the compiler proposed here could be used for the actual compilation process. Unfortunately, the Octopus tool does not currently support source code generation, and the final version is scheduled to support code generation for the Java platform, rather than C#.

As the Octopus tool was not released until June 1st 2004, our implementation was not able to benefit from it. Our implementation was developed independently from the Octopus tool, and the other contract tools presented in this section. Our implementation was completed in early April 2004[2]. We will turn our attention to the second tool, the object constraint language environment (OCLE).

2.5.2 The Object Constraint Language Environment

The object constraint language environment (OCLE) is a UML CASE tool that offers full OCL 2.0 support at both the UML metamodel and model levels[4]. The OCLE allows a UML model to be augmented with OCL expressions. Like the Octopus tool, OCLE works at the model level and is Java

¹⁷In the beta version.

based. The OCLE allows for importing of UML models via XMI, and then allows for augmentation with various OCL constraints. The OCLE supports code generation from the model and represents OCL constraints via Java code.

The OCLE is designed and maintained by Babes-Bolyai University in Romania. Version 2.0 of their tool, to support the new features of the OCL 2.0, was released in December 2003. Since then various maintenance releases have been issued.

The OCLE and the Octopus tool are the only two known tools that support version 2.0 of the OCL. Microsoft is developing a new language which adds constraints to the C# language. This new language is examined in the following section.

2.5.3 Spec#

Spec# is being developed at Microsoft Research by the software productivity tools group[27]. The goal behind Spec# is to add enforceable contracts to .NET[22]. Spec# provides an extension to C#, which contains non-null types, preconditions, postconditions, object invariants, and checked exceptions. Figure 2.57 illustrates an example of the Spec# syntax.

```
public virtual StringBuilder Append(char[] value, int startIndex, int charCount)
    requires 0 <= startIndex; otherwise ArgumentException;
{
}
```

Figure 2.57: *Spec# syntax*

Spec# works at the code level, and has no ties to models. The Spec# compiler inserts dynamic checks to enforce the contracts. Spec# is currently in development at Microsoft[21]. While Spec# allows for the insertion of contracts at the code level, it does not use a separate language, such as the OCL, to specify the contracts. The contracts are specified by using C# expressions. As such, the contracts will be tightly coupled to the implementation language, C#, and may not be useable at the model level. An examination of two other tools for adding constraints at the code level follows. The two tools are Jcontract and iContract.

2.5.4 Jcontract

Parasoft's Jcontract is a design by contract tool[37]. Jcontract compiles Java source code that contains contracts in the form of comments. The result of a successful compilation is byte code that contains checks to verify the specified contracts at runtime.

Jcontract consists of three modules. The first module is the actual compiler. The compiler generates .class files with extra code to check the contracts specified in the source at runtime. The second module is the runtime, which checks and monitors the contracts at runtime. The third and final module is the monitor, which shows the progress of the application and reports any contract violations[37]. Rather than provide a tool that integrates into existing tools, Parasoft created a complete toolkit.

As the contracts are specified in comment blocks, if the developer does not wish to use the contract checking mechanism they may simply use the regular Java compiler. The regular Java compiler will simply treat the contracts as comments.

As with Spec#, the syntax for the contract is based on the implementation language. If used at a higher level of abstraction, the contracts would have to undergo a significant transformation in order to be applied to multiple implementation technologies. And the contracts are placed in comments, they may interfere with actual source comments. We will now round out our examination of existing tools by looking at iContract.

2.5.5 iContract

Like Jcontract, iContract uses source comments to specify preconditions, postconditions, and class invariants in Java programs[20]. However, unlike Jcontract, iContract uses OCL 1.0 expressions to define the contracts. As stated before, the use of a separate language to define the contracts allows the contract definitions to be made regardless of the implementation technology. Then when the PIM is transformed to a PSM, the contracts are then simply promoted to the PSM level without a need for an implementation specific transformation. Figure 2.58 illustrates the syntax employed by iContract. Figure 2.58 defines a single precondition, which states that the parameter "f" must be greater than or equal to zero.

```
/*  
 * @pre f >= 0.0  
 */  
public float sqrt(float f) { ... }
```

Figure 2.58: Example of *iContract* syntax

As *iContract* also specifies contracts via comments, it has the same drawbacks as illustrated in the previous section. The driver for placing contracts in source comments is that the source can be compiled by a compiler that does not support contracts. With this fact in mind, our attention will focus on a language that is designed to support contracts as part of the language, rather than an add-on.

2.5.6 Eiffel

The Eiffel programming language is named after the engineer Gustave Eiffel, the creator of the famous Eiffel tower in 1887[42]. As the Eiffel tower was completed on time and within budget, the Eiffel language was designed to do the same with software projects. Eiffel was developed by ISE in 1985. Eiffel became a mainstream language in 1988 with the release of Bertrand Meyer’s seminal work on contracts[25]. Eiffel has gone through various revisions and upgrades. The current version, 2.01, comes complete with an advanced integrated development environment[42].

Eiffel is a fully object-oriented language, with a very verbose and English like syntax. Eiffel allows for the expression of analysis, design, and implementation all in the same language[42]. In Eiffel, all code must be placed within a class. A class is made up of features. A feature can be either an attribute, or a routine. An attribute is a data item. A routine can either be a function that returns a value, or a procedure that does not return a value. In Eiffel, everything is expressed in terms of contracts. Figure 2.59 shows a simple “counter” class written in Eiffel.

As illustrated in Figure 2.59, a precondition is expressed by using the *require* keyword, postconditions via the *ensure* keyword. Invariants are defined via the *invariant* keyword.

As all aspects of Eiffel are written in one language, there is no separation between the implementation and the contracts. In Eiffel the contracts make up the implementation. Our approach uses two different

```

indexing
  description: "Counters that you can increment by one,
  decrement, and reset"
class interface
  COUNTER
  feature -- Access
    item: INTEGER
    -- Counter's value.
  feature -- Element change
    increment is
      -- Increase counter by one.
    ensure
      count_increased: item = old item + 1
    decrement is
      -- Decrease counter by one.
    require
      count_not_zero: item > 0
    ensure
      count_decreased: item = old item - 1
    reset is
      -- Reset counter to zero.
    ensure
      counter_is_zero: item = 0
  invariant
    positive_count: item >= 0
end

```

Figure 2.59: *A counter class written in Eiffel*

technologies and integrates them to be used as a single technology, while still maintaining the separation between contracts and implementation. An alternative to the OCL, Alloy, is described below.

2.5.7 Alloy

Alloy is a formal constraint specification language[18]. Alloy can be used as an alternative to the OCL. Alloy is a simple, precise, and tractable notation for object modeling. Alloy allows for automatic analysis through its analysis tool, Aloga[28]. Aloga can perform simulations and consistency checking of constraints specified in Alloy.

Alloy has several differences over the OCL. The core differences are as follows. Scalar types are treated as singleton sets. This treatment allows for sets and scalars to be treated the same with respect to navigation[44]. Types are implicit and are associated with various domains. A domain is defined as a set that is not declared as a subset of any other set[44]. Classes correspond to sets that are subsets of the domains, and subclasses correspond to subsets[44].

Syntactically, an Alloy specification consists of one or more paragraphs. A paragraph can be one of two types. The first is a domain paragraph; which defines all domains used in the specification. The second type of paragraph is a state paragraph. State paragraphs declare additional sets whose elements come from the domain paragraphs. The declarations may include multiplicity constraints, mutability constraints, and partitions[17].

Figure 2.60 illustrates an example of an Alloy specification. The figure defines two domains “Person” and “Name”. The sets “Man” and “Woman” partition the “Person” set. This means a man cannot become a woman. The parents relation defines that a person’s parents do not change. The siblings relation is defined as being many-to-many, while the wife relation states: one man to at most one woman. Name relates a person to a unique name¹⁸. For more information on Alloy please see [17, 18, 28, 44]. The OCL was chosen over Alloy because of its wide spread usage. Developing a similar compiler for Alloy should not be any more difficult than the OCL version.

```
model Family {
  domain{Person, Name}
  state {
    partition Man, Woman: static Person
    Married: Person
    parents: Person -> static Person
    siblings: Person -> Person
    wife(~husband): Man? -> Woman?
    name: Person -> Name!
  }
  def siblings {
    // Two people have the same siblings if and only if
    // they have the same parents
    all a, b | a in b.siblings <-> (a.parents = b.parents)
  }
  inv Basics {
    // A person has a wife if and only if he is a man and is married
    all p | some p.wife <-> p in Man & Married
    // No persons wife is also a sibling
    no p | p.wife in p.siblings
    // A person has at most one father and at most one mother
    all p | (sole p.parents & Man) && (sole p.parents & Woman)
    // No person is an ancestor of him/herself
    No p | p in p.+parents
  }
}
```

Figure 2.60: Example contract specification using Alloy

¹⁸Figure taken from [17].

2.5.8 Summary

Table 2.3 provides a summary of the tools and technologies discussed in this section.

Tool / Technology	Constraint Language	Evaluation Level	Code Generation
The Octopus Tool	OCL 2.0	Model Only	No
The OCLE	OCL 2.0	Model Only	Java
Spec#	C#	Code Only	N/A
Jcontract	Java	Code Only	N/A
iContract	OCL 1.0	Code / Model	N/A
Eiffel	Eiffel	Code Only	N/A
Alloy	Alloy	Code Only	No

Table 2.3: *Summary of existing tools and technologies*

As shown by the diversity of the various tools and technologies related to software contracts, there are conflicting ideas of what software contracts are and how they should be implemented. With the development of Spec# and the continued progress of Eiffel, it should be evident that there is much active research and development in the field of software validation through the use of contracts.

2.6 Summary

Under the design by contract methodology; preconditions, postconditions, and class invariants can be used to place constraints on software systems. Preconditions specify constraints that must be true in order for a method to execute. Postconditions must be true following the execution of a method. Class invariants are applied at the class level, and must be true before and after the execution of each method contained within a class. If any type of constraint fails, an exception is generated.

The OCL acts as a plug-in to the UML to provide a formal constraint language that can be used in UML models. A constraint language is needed, as UML diagrams alone do not contain enough information to be used in the model transformation processes defined under MDA. UML diagrams alone are imprecise system descriptions, the OCL can be used to add the missing precision.

The Microsoft .NET Framework brings together a common set of APIs and class libraries for use with any .NET programming language. The framework allows for the interoperability of multiple program-

ming languages in the same application. The Microsoft .NET Framework defines a new .NET specific programming language, C#. C# is based on C++, with some notable exceptions. C# does not contain pointers, and has automatic garbage collection. C# also has several new program elements that do not exist in other programming languages. Attributes, properties, indexers, delegates, and events are some of these new program elements.

Various tools and technologies exist to add and specify software contracts. Research and development is still progressing to find new and improved mechanisms for the validation of software.

Chapter 3

Syntax

The first step in integrating the OCL into C# is to determine how the two languages will be expressed together. This chapter will examine how both the OCL and C# can be expressed together at the syntax level. It begins by introducing the notion of an OCL block. The OCL blocks are used to express OCL expressions directly in a C# source file. Following the definition of the OCL blocks, the necessary changes to the C# grammar will be explored. Once the C# grammar is modified to support the OCL blocks, we will examine the action of parsing the OCL. A discussion covering some of the modifications that were made to the stock OCL grammar to support various enhancements will follow. The chapter will conclude with a discussion of the process of matching an OCL expression with its corresponding C# element.

3.1 OCL Blocks

In order to allow for maximum flexibility with respect to the specification of the OCL at the code level, the notion of an OCL block has been defined. An OCL block consists of zero or more OCL expressions encompassed in notation indicating the start and end of the OCL block.

An OCL block is specified by using the *OCL* keyword to indicate the start of the block, followed by an opening square bracket. The end of the OCL block is denoted by a closing square bracket. The body of

the OCL block is located between the opening and closing square brackets. The body consists of zero or more C# string literals (zero or more quoted strings). If a string spans more than one line, each line must consist of a fully quoted string. Figure 3.1 illustrates an OCL block.

```
    OCL
    [
        "context Customer::Birthday() : OclVoid"
        "-- Some OCL Expression"
    ]
```

Figure 3.1: *Example of an OCL block with a single expression*

The OCL block shown in Figure 3.1 only contains a single OCL expression. It should be noted that an OCL block could contain any number of OCL expressions. Figure 3.2 illustrates an example of such an OCL block.

```
    OCL
    [
        "context Customer::Birthday() : OclVoid"
        "-- Some OCL expression"
        "-- Another OCL expression for Customer::Birthday"
        "context Customer::Transaction() : Integer"
        "-- An OCL expression for Customer::Transaction"
    ]
```

Figure 3.2: *Example of an OCL block with multiple expressions*

Semantically, the *OCL* keyword instructs the C# compiler that the information located between the square brackets is to be parsed via the OCL parser and not the C# parser. The quoted strings are concatenated and treated as a single string, which is then used by the OCL parser. Each string must be quoted so that the C# tokenizer does not convert the OCL expression into tokens, but rather preserves it as a literal string for the OCL tokenizer. Section 3.4 provides more information on the OCL parser. The C# grammar addition for parsing the OCL blocks is shown in Figure 3.3. The entire modified C# grammar is provided in Appendix C.

3.2 C# Grammar Changes

As different developers have different styles, the specification of the OCL blocks should be as natural as possible for the developer. In order to achieve this design goal, the OCL blocks should be allowed to be placed almost anywhere in the source code. If the developer wants to have all of the OCL blocks in a

```

ocl_expression:
    ocl-sections
ocl-sections:
    ocl-section
    ocl-sections ocl-section
ocl-section:
    OCL [ ocl-bodyopt]
ocl-body:
    string-literal
    ocl-body string-literal

```

Figure 3.3: C# grammar for parsing an OCL block

separate file or location, and have the C# code in a separate file or location, this should be possible. On the other hand, if the developer wants to intermix the OCL blocks with C# code in any order this should also be possible.

In order to accommodate this, the proposed modification to the C# grammar allows zero or more OCL blocks to be placed before any structural C# programming element. That is, the OCL blocks can be placed before the following C# programming elements: namespace declarations, using directives, attributes, classes, structures, interfaces, methods, operators, constructors, destructors, properties, accessors, indexers, fields and constants. In short, the only location where the OCL blocks are not allowed is in behavioral elements, such as method bodies. Figure 3.4 illustrates both correct and incorrect placement of OCL blocks.

```

class Customer
{
    int age = 0;
    // Correct OCL block placement
    OCL
    [
        "context Customer::Birthday() : OclVoid"
        "post: age = age@pre + 1"
    ]
    public void Birthday()
    {
        // Illegal OCL block placement
        OCL
        [
            "context Customer::Birthday() : OclVoid"
            "post: age = age@pre + 1"
        ]
        age++;
    }
}

```

Figure 3.4: Placement of OCL blocks

In Figure 3.4, the correct OCL block is placed immediately prior to the method for which the constraint is defined. In the case where the OCL blocks are placed in a separate file, the OCL blocks may not be located immediately prior to the constraint target. In order to accommodate the random locations of the OCL blocks, there is no restriction where the OCL blocks are defined with respect to their actual constraint target. Conversely, every OCL block must be started with a fully qualified context declaration. The context declaration is required even if the constraint target is implicit. An OCL expression that does not contain a fully qualified context declaration is considered invalid.

As the context declaration is part of the OCL expression, it cannot be determined at discovery time. That is, the C# parser is not able to match the OCL blocks with their target at parse time. The decision must be postponed until the C# parser has finished discovering all of the types defined in the system, and the OCL block is parsed to extract the information from the context declaration portion of the OCL expression.

As the target for a given OCL expression cannot be determined until later, the OCL expression is attached to the structural item immediately after the definition of the OCL block that the expression is contained in. Figure 3.5 illustrates this. Figure 3.5 shows an OCL block that contains an expression that should be assigned to the “GetAge” method located in the “Person” class. However, as the OCL expression has not yet been parsed, and it is possible that the C# parser has not yet encountered the “Person” class, the OCL expression is assigned to the “Customer” class until the “Person” class can be found and the OCL can be parsed.

```

OCL
[
    "context Person::GetAge() : Integer"
    "post: result = self.age"
]
Class Customer
{
}

```

Figure 3.5: Assignment of OCL blocks

The full modification to the C# grammar to support the OCL blocks as discussed in this section is located in Appendix C. The actual grammar modifications are rather simple. The modifications consist of adding a new keyword to the C# language: *OCL*, and adding the *ocl_expression* symbol to the beginning of the definitions for all the structural elements.

The notable disadvantage of the OCL block notation is that a normal C# compiler will not be able to compile the files, as the OCL is not contained within comments. If the use of design by contract through the OCL is desired, a specialized compiler will be required. The specialized compiler outlined in this thesis can be used to support design by contract with the OCL. The compiler contains various compile-time options, which can enable or disable the OCL constraints as needed. Appendix B contains colour plates showing the specialized compiler. The completed implementation can be downloaded from: [http://www.ewesimplex.ca/csocl\[2\]](http://www.ewesimplex.ca/csocl[2]).

In the case where the use of a non-OCL compiler is required, the OCL blocks can be removed by using a pre-processing step. The pre-processing step would locate the OCL blocks and either comment them out via C# comments or remove the blocks altogether.

An alternative to the OCL block notation would be through the specification of OCL constraints through the use of C# attributes. The information contained within an OCL attribute could then be fed into the OCL parser. An advantage to the use of an OCL attribute is that a non-OCL compiler could be used to compile the source files with no modification. The disadvantage is that the OCL attributes would have to be placed directly on the corresponding C# element. In order to allow for maximum flexibility, and to allow for a separation between the C# language and the OCL expressions, the notion of an OCL block is used in the implementation. The OCL block placement strategy provides the developer with flexibility on where to place the OCL expressions. This flexibility will provide help cater to different programming styles and methodologies.

The modifications to the C# grammar are complete. The OCL expressions located in the OCL blocks have been converted into a single string and associated with the C# structural element located immediately following the OCL block. The next step is to parse the OCL expressions to determine the correct context declaration for each expression. Parsing the OCL expressions is the focus of the next section.

3.3 Parsing OCL Expressions

As stated in the previous section, we now need to parse the OCL expressions to determine the correct context declaration. Structurally, the C# has been parsed into an abstract syntax tree (AST). Each

structural element in the AST will have zero or more OCL expressions, each in a single string. It should be noted that the AST has not been semantically analyzed by the compiler.

Each OCL expression is parsed independently from the others. The result of a successful parse is an AST representing the structure for each OCL expression. Each of the OCL expressions, which were represented by a string, is now represented by an AST dedicated to a single OCL expression. The next section will examine the grammar used to parse the OCL.

3.4 The OCL Grammar

The OCL specification defines a grammar which is partially suitable for parsing[32]. The main issue in the grammar is that it is ambiguous unless the model to which the OCL is being applied is available. Meaning, an OCL expression can only be parsed if the model to which the expression is being applied to is available for use by the parser. As a parser generator tool, JAY[47], is being used to generate the actual parser, the parser will not be able to reference the model. Several changes have been made to the grammar in order to allow it to be used in the context of a parser generator, and without the need for a reference model during the parsing phase. While the majority of these changes are syntactic in nature, the OCL related changes are outlined in the next sections. The complete JAY ready grammar can be found in Appendix D. Figure 3.6¹ illustrates each of the grammatical symbols that make up the OCL grammar in a UML class diagram.

3.4.1 Variable Expressions

A variable expression consists of a path name. A path name is defined as one or more simple names, which are connected via the scope operator. A simple name is a sequence of characters, or an identifier. For example, a valid path name would be “Data::Customer”. A path name can also represent an enumeration literal, or an attribute call expression. The only way to determine if a path name is a variable

¹Aggregations were removed for readability.

Figure 3.6: UML class diagram showing the OCL grammatical elements

expression, an enumeration literal, or an attribute call expression, is to break up the path name into sections and use the model to determine the correct expression type. As the model is not available at parse time, all path names will be resolved into variable expressions and their meaning will be determined during the semantic pass.

3.4.2 Iterator Variable Lists

The OCL specification[32] states that if an iterator expression contains more than one iterator variable, then the variables are to be separated by a comma². If the comma is used, then the parser cannot determine the difference between a method call with a parameter list and an iterator expression. As the expressions have different structures, the method used in the previous section cannot be used here. In the previous section the resolution of the symbol was deferred until the semantic pass. The delayed resolution cannot be used because, regardless of the expression type, method call or iterator, the syntactical structure of the expression is the same. In this case, the structure of a method call and an iterator expression is not the same. In order to resolve the ambiguity, multiple variables in an iterator expression will be separated by using semicolons, as illustrated in Figure 3.7. The iterator in Figure 3.7 defines a set containing three numbers, and then uses the default OCL iterator to compute the sum of the numbers in the set.

```

OCL
[
  "context IterateTests::Iterate1() : OclVoid"
  "pre: Set {1, 2, 3}->iterate(i : Integer;
    sum: Integer = 0 | sum + i) = 6"
]

```

Figure 3.7: A modified iterator expression

3.4.3 Iterators as Methods

The grammar defined in the OCL specification[32] allows methods to be called using both the “.” and the “->” operators. The grammar also allows iterators to be called using both operators as well. As both

²The term iterator expression is in reference to both the generic iterate and the stock iterator expressions.

types of expressions can be used in both contexts, the parser is not always able to distinguish between the iterators and the operations. In order to correct this, we decided that iterators can only be used via the “->” operator. Methods can still be called using both operators, as they have a different semantics.

3.4.4 Properties

The standard OCL specification[32] supports specifying preconditions, postconditions and body rules for all of the C# elements, except for properties and indexers. Properties and indexers do not contain ending parenthesis, and thus do not follow the OCL notation for a method. The existing OCL grammar only allows for preconditions, postconditions, and body rules to be specified on elements that contain parenthesis.

The specification of preconditions, postconditions and body rules is a vital requirement for integrating the OCL with C#. In order to allow preconditions, postconditions and body rules to be specified on properties, a modification is required to the OCL grammar. The modification allows an attribute specification in the context declaration to be either followed by the *get* or *set* keyword. If the declaration is followed by the *get* keyword, the compiler will attach the preceding OCL expression with the *get* method of the named property. If the declaration is followed by the *set* keyword, the compiler will attach the proceeding OCL expression with the *set* method of the named property. Figure 3.8 shows an example of applying an OCL precondition and postcondition to a property.

It should be noted that even if constraints are specified on the *set* accessor of the “Age” property in Figure 3.8, the property type must still be specified. The property type is required on both accessors, so that the correct accessor can be found to assign the OCL constraints to. With support for C# properties added to the OCL grammar, the same can now be done for indexers.

3.4.5 Indexers

As with properties, indexers do not contain ending parenthesis. Using the same arguments as in the previous section, indexers should also enjoy the use of preconditions, postconditions and body rules. In

```

class Person
{
  OCL
  [
    "context Person::Age set : Integer"
    "pre: value > 24"
    "post: age = value"
  ]
  OCL
  [
    "context Person::Age get : Integer"
    "post: result = age"
  ]
  public int Age
  {
    get { return age; }
    set { age = value; }
  }
}

```

Figure 3.8: OCL that allows constraints to be applied to C# properties

order to facilitate this addition, an additional OCL grammar modification is required; the addition of the *indexer* keyword to the OCL keyword list. The *indexer* keyword follows the class name to which the indexer is being applied. The *indexer* keyword is then followed by the indexer's parameter list, and then either a *get* or a *set* keyword to indicate which of the indexer's accessors the expression is being applied to. Figure 3.9 shows the use of the *indexer* keyword.

```

class Test
{
  OCL
  [
    "context Test indexer(i : Integer) get: Integer"
    "post: result = i * i"
  ]
  public int this[int i]
  {
    get { return i * i; }
  }
}

```

Figure 3.9: Use of the *indexer* keyword

The addition of the *indexer*, *get*, and *set* keywords is not part of the OCL specification[32]. These keywords and their semantics are added in order to provide a more robust integration of the OCL with C#. If the standard OCL specification is desired, the OCL extensions can be turned off via the use of a compiler option.

3.4.6 Summary

With the grammar modifications complete, we can now parse the OCL expressions without model knowledge, into an AST. The AST contains a few symbols that will need to be resolved further during the semantic pass, but the majority of the symbols and all the syntactic errors are caught by the parsing process. A complete list of the errors generated by the specialized compiler that are related to the OCL can be found in Appendix E, which contains both syntactic and semantic errors.

3.5 The Assignment of OCL Expressions

As the parsed OCL expressions may not be assigned to the correct C# element, we need to move them to the correct C# element as needed. This section will discuss the process required to complete this task.

As illustrated in the previous sections, each OCL expression can be divided into three pieces. The first is the context declaration. The second is a label indicating how the OCL expression is to be used. The third, and final piece, is the actual OCL expression. As all of the ambiguous grammatical issues reside in the third portion, they will not be discussed until Chapter 5. If we take Figure 3.9 as an example, the first line of the OCL expression is the context declaration. The second line contains both the label, post, and the actual OCL expression.

In order to determine which C# element that a given OCL expression belongs to, we only need to examine the context declaration portion. There are three categories of context declarations: association, classifier, and operation. Each category of context declaration will be discussed in the following sections.

3.5.1 Association Context Declarations

Association context declarations are used to assign either initialization or derivation rules to attributes. In C#, the notion of an attribute is in reference to either a field or a constant. Figure 3.10 illustrates an example of an association context declaration.

```
context Init::_bool1 : Boolean
```

Figure 3.10: *Example of an association context declaration*

An association context declaration consists of the *context* keyword, followed by a path name, followed by an optional type. The *context* keyword simply marks the beginning of the context declaration. The path name should be a fully qualified path. The path should indicate the classifier name, followed by the attribute name. The type specified indicates the type of the attribute.

The first step in locating the correct C# attribute is to split the path name into the classifier name and the attribute name. The classifier name is everything except for the last name. In Figure 3.10, the classifier name is “Init”, and the attribute name is “_bool1”. If the classifier is located in a namespace, the namespace name should precede the classifier name. With the classifier name, the compiler will then look for a type that matches the classifier name. A compile-time error will be generated if the type cannot be found.

With the matching classifier located, the next step is to determine if the corresponding attribute is defined on the classifier. If an attribute is located, the types are checked. If the type is not specified on the context declaration, then no type check is performed. In the case that the label portion of the OCL expression defines an initialization rule, the actual OCL expression defining the initialization rule is assigned to the field. As the field has been located and the actual OCL expression assigned, the context information is discarded³. In the case where a derive rule is used, the process is more complex. An initialization rule simply defines the initial value for an existing attribute. A derive rule specifies an expression that defines the value of the field. In C#, a field contains a single value and thus cannot be represented by an expression. In order to allow a field to be used as a derived value, the field is replaced by a read-only property. If the label portion of the OCL expression defines a derive rule, then the field is removed from the classifier, and replaced by a read-only property with the same name, and of the same type as the removed field. The actual OCL expression that makes up the derive rule is assigned to the get accessor of the new property. The new property does not contain any code at this point. Only the derive OCL expression is assigned to the get accessor. The code will be generated during the semantic pass.

³Discarded means that the context information is not needed by the compiler. The context information is still stored for debugging and exception raising purposes.

If a field cannot be found in the classifier, the constants, if any, are examined. If a constant can be found, and the actual OCL expression defines an initialization rule. The initialization rule is then assigned to the constant. If the OCL expression defines a derive rule, then an error is generated, as a derive rule may specify an non-constant expression.

If neither a field nor a constant can be located in the classifier, and the actual OCL expression is an initialization rule, then an error is generated. If, however, the actual OCL expression is a derive rule, then a new read-only property is created and the actual OCL expression is assigned to the get accessor of the new property.

While it is correct for both an initialization rule and a derivation rule to be specified on the same attribute, the derivation rule will be used to define the attribute and the initialization expression will be ignored. The initialization expression will be ignored as the attribute's value will be calculated via the derivation rule. With expressions defined on attributes handled, we will now examine the second category of context declarations.

3.5.2 Classifier Context Declarations

Classifier context declarations are used to specify either class invariants or definition rules. Figure 3.11 illustrates a classifier context declaration. A classifier context declaration consists of the *context* keyword, followed by the fully qualified classifier name. As with association context declarations, the classifier name may be prefixed with namespace names as needed.

```
context InvTests
```

Figure 3.11: *An example of a classifier context declaration*

The first step in locating the correct C# element is to locate the classifier that is referenced in the context declaration. If the classifier cannot be located, a compile-time error is generated. If the actual OCL expression is a class invariant, then the OCL expression that defines the class invariant is assigned to the classifier. If the actual OCL expression is not a class invariant then it must be a definition rule.

A definition rule may define either a new attribute or a new method. If a new attribute is defined, the compiler first checks to make sure that an attribute or property with the same name as the new one does

not already exist. If so, a compile-time error is generated. Using the same rationale as with derivation expressions, a new read-only property is created for the definition rule. The actual OCL expression that defines the rule is assigned to the get accessor of the new property.

If the definition rule defines a new method, then the compiler checks to see if a method with the same signature as the one defined via the definition rule already exists. If one does, then a compile-time error is generated. Once it has been verified that the definition rule is defining a new method, the new method is created in the corresponding classifier. The body of the method is undefined; however, the actual OCL expression that defines the body of the method is attached to the method, and will be used to generate the method's body during the semantic pass.

With classifier context declarations handled, we will now examine the third and final category of context declarations; operation context declarations.

3.5.3 Operation Context Declarations

Operation context declarations are used to specify preconditions, postconditions and body expressions on methods, properties, and indexers. Figure 3.12 provides an example of an operation context declaration.

```
context Customer::Test3(a : Integer, b : Integer) : Integer
```

Figure 3.12: *An example of an operation context declaration*

An operation context declaration is defined by the *context* keyword followed by a path name. The path name is followed by parenthesis and an optional parameter list. Finally the return type of the operation rounds out the operation context declaration. The path name consists of a fully qualified classifier and the operation name. The optional parameter list consists of an ordered list of parameters, each separated by a comma. The parameter type is optional on preconditions and postconditions, but must exist for body rules. The return type of the operation is also optional on preconditions and postconditions.

The first step in locating the correct C# element, is to separate the path name into a classifier portion and the operation name portion. From Figure 3.12, the classifier name would be “Customer”, and the

operation name portion would be “Test3”. The compiler then attempts to locate the classifier specified by classifier portion of the class name. If a classifier cannot be located, a compile-time error is generated.

With the classifier located there are two cases. The first case is that preconditions and/or postconditions are being added to an existing operation. The second case is that a body rule is being used to define a new operation.

In the case where preconditions and/or postconditions are being added to an existing operation, the compiler searches through the properties, indexers, constructors, and methods of the classifier to find a match. If a match is found, then each actual OCL expression that makes up a precondition or postcondition is assigned to the corresponding operation.

In the case where a body rule is being used to define a new operation, the compiler searches through the properties, indexers, constructors, and methods of the classifier to make sure that there are no existing operations that match the one being defined via the body rule. If such a match exists, a compile-time error is generated. If no match is found, then the corresponding operation is created, and the body rule is assigned to the operation. Like definition and derivation rules, the new operation does not contain a body. The body will be generated when the actual OCL expression defined by the body rule is resolved during the semantic pass. If there are preconditions and/or postconditions that go with the body rule, they are also assigned to the new operation.

With the three categories of context definition expressions handled, all of the OCL expressions have been moved from their temporary location, and assigned to the correct C# element that the context expression references.

3.6 Summary

OCL expressions are added to C# source code using OCL blocks. The OCL blocks can be placed in any location with the exception of inside behavioral elements. The required modifications to the C# grammar to support the use of OCL blocks have been completed. The modified C# grammar is located in Appendix C.

The OCL grammar provided in the specification[32] contains various ambiguities, including the handling of variable expressions, variable lists in iterators, and operation expressions overlapping with iterator expressions. The proposed grammar corrects these ambiguities and adds support for adding OCL expressions to two unique elements found in C#: indexers and properties.

With the OCL expressions parsed and the action of assigning the actual OCL expression to the C# element, the context declaration processing is complete. The next step is to perform the semantic pass on the OCL expressions, and resolve each of them into C# code. Before, the semantic pass is discussed in Chapters 5 and 6, Chapter 4 will define what it means to place a constraint on each of the C# structural elements.

Chapter 4

Semantics

With the parsing of the OCL expressions complete, we now have each of the OCL expressions assigned to their corresponding C# element. The assignment is represented by a link, the OCL expression's AST has not been integrated into the main C# AST. The integration process is presented in Chapters 5 and 6. This chapter will examine what it actually means to apply an OCL constraint to a C# element, and the effect that the constraint has on that element. The chapter will begin by looking at exactly which C# elements can have an OCL precondition, postcondition or class invariant. Following the discussion of constraints, each C# program element is examined, and the effect of placing a constraint on that element is explored.

The semantics discussed in this chapter represent a complete picture of the meaning that an OCL expression could have when applied to a given C# element. Due to the complexity of some semantics, not all of the topics presented in this chapter have been implemented. Footnotes will be used to indicate such elements.

4.1 Preconditions and Postconditions

As stated previously, preconditions and postconditions can be placed on any C# method, including constructors and destructors. The compiler maintains two separate lists for each method: one for the

preconditions, and one for the postconditions.

Each of the preconditions and postconditions are expressed as boolean expressions. If the constraint cannot be evaluated into an expression of type Boolean, an error is generated. The boolean expressions are stored in their corresponding list.

During the code generation phase, the boolean expressions that correspond to the preconditions will be inserted at the beginning of the method along with exception raising code. The exception raising code will execute only if one or more of the boolean expressions fail. The boolean expressions that correspond to the postconditions will be appended to the end of the method, along with exception raising code that is similar to that of the preconditions. Figure 4.1 illustrates a method augmented with preconditions and postconditions. For the precise technical information on the code generation process, see Chapter 6.

```
void Method()
{
    if(!preconditions)
    {
        RaiseException();
    }
    // Original method Body.
    if(!postconditions)
    {
        RaiseException();
    }
}
```

Figure 4.1: *Method augmented with preconditions and postconditions*

In the case where more than one precondition or postcondition exists for a given method, the *if* statement consists of all the condition's boolean expressions "and"ed together. Each of the conditions must evaluate to true for the preconditions or postconditions to be valid. If one or more fails, an exception is raised.

4.2 Class Invariants

As previously stated, class invariants can be placed on any structure or class. Each class and structure maintains a list of all the invariants that have been assigned to it.

Class invariants are checked on the entry and exit of every method. The class invariants are checked immediately before the preconditions, and immediately following the postconditions. Unlike preconditions and postconditions, class invariants are not checked on entry to constructors, or exit from destructors. By not having the class invariants checked on entry to a constructor, the constructor can set default values for attributes that may be used in various class invariants. Likewise, by not having invariants checked on exit from a destructor, the destructor is able to invalidate the class invariants by freeing resources used by the attributes.

Each of the class invariants are resolved into a boolean expression. The boolean expressions are stored as a list at the class or structure level. During the code generation phase, when code for a method is being generated, the boolean expressions are processed by the method's code generator and the corresponding class invariant testing code is added to the method's code block. Figure 4.2 augments Figure 4.1, with class invariants added. The technical details of the code generation process for class invariants will be examined in Chapter 6.

```
void Method()
{
    if(!classInvariants)
    {
        RaiseException();
    }
    if(!preconditions)
    {
        RaiseException();
    }
    // Original method Body.
    if(!postconditions)
    {
        RaiseException();
    }
    if(!classInvariants)
    {
        RaiseException();
    }
}
```

Figure 4.2: *Method augmented with preconditions, postconditions and class invariants*

4.3 Constraints on Properties

A property consists of at most two accessors: a get accessor and a set accessor. Each accessor is defined as a series of statements. This allows each accessor to be seen as a method at the contract level. Preconditions and postconditions can be added to accessors contained within properties. The preconditions and postconditions have the same behavior as discussed in Section 4.1.

As there is no restriction on the functionality of the statements contained within an accessor, it is possible that the class invariants will become invalidated. To this end, class invariants are checked on the entry and exit from all property accessors. Class invariants located in accessors have the same behavior as discussed in Section 4.2.

4.4 Constraints on Indexers

Like properties, indexers consist of at most two accessors: a get accessor and a set accessor. Each accessor is defined as a series of statements. The accessors that are contained in indexers are the same as the accessors contained in properties. Preconditions, postconditions, and class invariants are processed as stated in the section on properties.

4.5 Constraints and Delegates

Preconditions and postconditions can be placed on delegate declarations. If a delegate declaration contains preconditions and/or postconditions, the conditions will be assigned to any method that is referenced by that delegate. If the method is called via the delegate, then the constraints specified via the delegate declaration will be enforced. If the same method is called outside of the delegate, then the constraints specified via the delegate declaration will not be enforced. This means that the runtime environment determines the context of the method call. If the method is invoked via a delegate, then the delegate's constraints are executed along with any constraints placed on the method itself. If the method

is invoked outside of the delegate, then only the constraints placed on the method itself are executed.

Figure 4.3 illustrates how preconditions and postconditions are used in delegates.

```

delegate void MyDelegate(int i);
class Test
{
    public static void M1(int i)
    {
        if(invokedViaDelegate)
        {
            if(!delegatePreconditions)
            {
                RaiseException();
            }
        }
        if(!M1Preconditions)
        {
            RaiseException();
        }
        // M1 Body
        if(!M1Postconditions)
        {
            RaiseException();
        }
        if(invokedViaDelegate)
        {
            if(!delegatePostconditions)
            {
                RaiseException();
            }
        }
    }
    public static void Main()
    {
        MyDelegate d = new MyDelegate(Test.M1);
        d(1);
    }
}

```

Figure 4.3: *Preconditions and postconditions used in the context of a delegate*

As shown in Figure 4.3, the preconditions and postconditions defined on the delegate are only executed if the method is called using the delegate. Various approaches can be used to determine how a method is invoked. One approach is to use a specialized delegate class that sets a flag that can be tested in the method to determine if the delegate was used to invoke the method. Another approach is to examine the stack trace to determine the caller of the method¹. Allowing for preconditions and postconditions to be placed at the delegate level provides for the ability to add constraints to a family of methods. In addition, an object that invokes a delegate can ensure that every delegate method fulfills a common set

¹Implementation Note: The semantics described in this section are not implemented in our compiler.

of constraints.

A delegate may be defined as a high-level entity, or in a class or structure that does not contain the method that is being called by the delegate. Thus, class invariants cannot be defined at the delegate level. Furthermore, if a delegate is defined in a class or structure that has class invariants, the class invariants do not propagate to the delegate or to the methods that are being invoked via the delegate. Class invariants are applied to methods that are invoked via a delegate, but only if the containing class or structure of the method (not the delegate) contains class invariants. Class invariants are applied to the method regardless if used by a delegate or not. The delegate itself does not contain or use class invariants.

4.6 Constraints and Events

As stated in Chapter 2, events maintain a list of event handlers (delegates). Constraints cannot be specified on events. If a constraint is required on the corresponding event handlers, the constraint may be added at the delegate level². As previously stated, placing constraints on a delegate allows for a set of constraints to be consistent for all operations called via that delegate. In the case of an event, all event handlers would have a set of constraints assigned to them via the defining delegate.

Events use the “+=” and “-=” operators to add and remove delegates from the event handler list. The behavior of the “+=” and “-=” operators can be defined by specifying add and remove accessors for the event as shown in Figure 4.4. The accessors have the same characteristics as the accessors used in properties and indexers. To that effect, preconditions and postconditions can be used to specify constraints on the add and remove operations of a given event. In addition, if class invariants exist on the container of the event, they will also be applied to the user-defined accessors. In the case of Figure 4.4, if a class invariant was defined on the “Button” class, it would be tested on the entry and exit from the add and remove accessors. If no accessors are specified by the user, class invariants will not be applied to the default accessors. The rationale, is that the default accessors are implemented by the runtime and not by the compiler, thus there is no way that a class invariant could be violated.

²Subject to delegate implementation support.


```
public class Button
{
    private EventHandler handler;
    public event EventHandler Click
    {
        add { handler += value; }
        remove { handler -= value; }
    }
}
```

Figure 4.4: *Illustration of the add and remove event accessors*

4.7 Constraints and Inheritance

As stated in Chapter 2, the standard way to handle contract inheritance is defined informally by Meyer in his work on contracts[25]. Binder then formalized Meyer's work by defining the Percolation pattern[5]. Recent work by Robert Findler[8] has shown that Meyer's method for handling contract inheritance does not always assign blame to the correct entity.

Meyer's method for handling contract inheritance works as follows. If a method is redefined in a subclass, the following rules apply for preconditions, postconditions and class invariants.

1. A precondition may be weakened, not strengthened, in a redefinition of a method in a subclass[45].
2. A postcondition may be strengthened, not weakened, in a redefinition of a method in a subclass[45].
3. An invariant for a superclass is inherited by its subclasses. A subclass may strengthen the invariant but cannot weaken it[45].

The following sections will examine the differences between the methods proposed by Meyer and Findler in the context of class invariants, preconditions and postconditions.

4.7.1 Class Invariants

Class invariants are the easy case. The methods proposed by Meyer and by Findler are in agreement with respect to class invariants. If a subclass contains class invariants, the invariants are propagated to each of the subclasses. In effect, it is as if the invariants were not specified at the superclass level, but

rather at the subclass level. From the implementation point of view, all the class invariants are converted into boolean expressions, and “and”ed together to determine if the invariants are valid or not. Figure 4.5 illustrates how invariants are handled with respect to inheritance.

```

OCL
[
  "context C1"
  "inv I1: ..."
]
class C1
{
  public virtual void M1()
  {
    if(!I1)
    {
      RaiseException();
    }
    // M1 Body
    if(!I1)
    {
      RaiseException();
    }
  }
}
OCL
[
  "context C2"
  "inv I2:..."
]
class C2 : C1
{
  public override void M1()
  {
    if(!(I1 && I2))
    {
      RaiseException();
    }
    // M1 Body
    if(!(I1 && I2))
    {
      RaiseException();
    }
  }
}

```

Figure 4.5: *Inheritance of class invariants*

As Figure 4.5 shows, the invariant “I1” defined in “C1”, is “and”ed with invariant “I2” defined in “C2”. If “I1”, “I2”, or “I1” and “I2” are false, an exception will be raised. We will now look at a case where Meyer and Findler have different approaches to handling contract inheritance: preconditions and postconditions.

4.7.2 Preconditions

Under inheritance, preconditions can be weakened, not strengthened. Under Meyer’s proposal, if a subclass weakens a precondition, the superclass precondition will still be true, creating a true precondition. If however, a subclass strengthens a precondition, the superclass’ precondition will generate a false value. Herein is the problem with Meyer’s method for inheritance. As we have already stated, when a precondition fails, the fault lies in the invoker of the method. However, in the case where the subclass strengthens the precondition, the fault is not with the caller, but rather with the subclass itself. Under Meyer’s method, the false precondition will always result in blame being assigned to the caller. Blame should be assigned to the subclass, not the caller, because the subclass strengthened the precondition rather than weakening it. Figure 4.6 illustrates this point.

```

class C1
{
    OCL
    [
        "context C1::M1(i: Integer): Integer"
        "pre: i > 2"
    ]
    public virtual int M1(int i)
    {
    }
}
class C2 : C1
{
    OCL
    [
        "context C2::M1(i: Integer): Integer"
        "pre: i > 10"
    ]
    public override int M1(int i)
    {
    }
}

```

Figure 4.6: *Meyer’s method for the inheritance of preconditions*

In Figure 4.6, if the caller to “C2::M1” uses a value of “i = 5”, the caller is not to blame because the caller has fulfilled its part of the contract. The precondition stated in “C1” states that “i” must be greater than 2. However, under Meyer’s method an exception will be raised and will place blame on the caller because “i” is not greater than 10. Findler’s method corrects this flaw, by allowing blame to be assigned to “C2” itself. In the case of Figure 4.6, there are four cases for the preconditions for “C2::M1”.

1. “C1::M1” is true, and “C2::M1” is true ($i = 11$) - In this case there are no violation and execution is allowed to continue.
2. “C1::M1” is false, and “C2::M1” is false ($i = 1$) - In this case there is a violation at both levels, and the caller is to blame.
3. “C1::M1” is true, and “C2::M1” is false ($i = 5$) - In this case there is a violation in the class hierarchy “C2” has strengthened the precondition, and thus “C2” is to blame not the caller.
4. “C1::M1” is false, and “C2::M1” is true (no such i) - In this case there is no violation, unless “C2” is being used as an instance of “C1”, in which case the caller is to blame.

As shown in the preceding cases, Findler’s method is able to correctly assign blame in all four cases. Meyer’s method will incorrectly assign blame in case three and could incorrectly assign blame in case four (depending on the context).

4.7.3 Postconditions

Under inheritance, postconditions can be strengthened, but not weakened. Under Meyer’s method, if a subclass strengthens a postcondition, the postcondition defined in the superclass will also hold, and thus the postcondition will be true. If a subclass weakens a postcondition, the postcondition defined in the superclass will no longer be true, and the postcondition will fail. In this case, blame will be assigned to a method in the subclass. However, it is not the method’s fault but rather the subclass itself (Figure 4.7).

In Figure 4.7, if the caller to “C4::M1” uses a value of “ $i = 2$ ”, the method is not to blame because the method has fulfilled its part of the contract because the postcondition stated in “C4” states that “ $i*i$ ” must be greater than 2. Under Meyer’s method, an exception will be raised and will place blame on the method because “ $i*i$ ” is not greater than 4. Findler’s method corrects this flaw by allowing blame to be assigned to “C4” itself. In the case of Figure 4.7, there are four cases for the postconditions of “C4::M1”.

1. “C3::M1” is true, and “C4::M1” is true ($i = 10$) - In this case there are no violations and the execution is allowed to continue.

```

class C3
{
    OCL
    [
        "context C3::M1(i: Integer): Integer"
        "post: result > 4"
    ]
    public virtual int M1(int i)
    {
        return i*i;
    }
}
class C4 : C3
{
    OCL
    [
        "context C4::M1(i: Integer): Integer"
        "post: result > 8"
    ]
    public override int M1(int i)
    {
        return i*i;
    }
}

```

Figure 4.7: *Meyer's method for the inheritance of postconditions*

2. "C3::M1" is false, and "C4::M1" is false ($i = 1$) - In this case there is a violation at both levels, and the method is to blame.
3. "C3::M1" is true, and "C4::M1" is false (no such i) - In this case there is a violation in the subclass' method, and blame is assigned to the method ("C4::M1").
4. "C3::M1" is false, and "C4::M1" is true ($i = 6$) - In this case there is a violation in the class hierarchy "C4" has weakened the postcondition, and thus "C4" is to blame not the method.

4.7.4 Summary of Findler's Method

As Findler's method allows for improved assignment of blame, it will be used in the implementation of the compiler. This section will provide a summary of the information presented in the previous sections, and will explain how Findler's method will be used in the context of the implementation discussed in Chapter 6.

Flat Contract Checking

Flat contract checking is the simplest. It occurs when there is no inheritance at all. Only constraints local to the current class are involved. In this case, blame can only be assigned to the caller (failed precondition), or to the method itself (failed postcondition).

Interface Implementation

Under interface implementation, a class (C) realizes an interface (I). When a method in C is called, its precondition must be checked. However, the precondition to be checked depends on the static type of the reference to the object that is invoked. In this case, that type can be either C or I. In addition, where instances of C are substitutable in locations where I is expected, the inheritance hierarchy must also be checked. In this case, I's preconditions must imply C's preconditions, and C's postconditions must imply I's postconditions. The specifics of these implications were discussed in Section 4.7.2 and Section 4.7.3 respectively.

In the case where more than one interface is implemented by a class, the interfaces are treated as a single interface (from the contract point of view).

Class Inheritance

Class inheritance uses the same logic as presented in the previous section. The only additional remark is that the conditions of every interface and every superclass of the originally instantiated class are checked at each method call and each method return to ensure that the inheritance hierarchy is valid. The conditions are checked under the rules presented by Findler[8]. As C# does not support multiple inheritance, the case where more than one class is inherited does not occur.

4.8 Constraints on Interfaces

Interfaces in C# do not allow for the specification of attributes. As class invariants mostly reference attributes, class invariant declarations are not allowed at the interface level.

Preconditions and postconditions can be specified on an interface's methods, properties, and indexers. The preconditions and postconditions that are specified at the interface level are applied to all of the realizations of a given interface using the rules presented in Section 4.1.

4.9 Constraints on Structures

Structures do not support inheritance, initializers, destructors or instance constructors that do not explicitly initialize each member. Constraints on structures behave the same as constraints on classes. As structures are value types, special care needs to be taken when copying and referencing structure values in OCL constraints at the implementation level.

In order to preserve structures, the checking of constraints on a structure is done via a utility class. This means that the runtime code needed to verify a given constraint on a structure is placed into a static operation of a dedicated utility class. The main rationale for this design decision is that structures are stored on the stack, and it is possible that the runtime code for checking a given constraint will introduce extra attributes in the structure³.

A secondary motivation is that structures are often used for interoperability between managed and unmanaged code. The interoperability depends on a structure signature[26, 46]. Adding constraint-checking code may modify the signature, and cause the interoperability to fail. Figure 4.8 illustrates the use of a structure for an interoperability operation. For more information on structures and interoperability, please see [26].

³Implementation Note: As constraints on structures require special handling, they are not implemented in the compiler described in Chapters 5 and 6.

```

/// <summary>
/// Structure to represent a Win32 POINT structure
/// </summary>
[StructLayout(LayoutKind.Sequential)]
public struct POINT
{
    public int x;
    public int y;
}
public class Exports
{
    /// <summary>
    /// ScreenToClient Export
    /// </summary>
    [DllImport("User32.dll", CharSet=CharSet.Auto)]
    public static extern bool ScreenToClient(IntPtr hWnd,
        ref POINT pt);
}

```

Figure 4.8: *Using a structure for interoperability between managed and unmanaged code*

4.10 Constraints on Abstract Classes

In terms of the OCL, the abstract classes found in C# are treated as normal classes. All constraints that can be placed on a non-abstract class can be placed on an abstract class. If a precondition or postcondition is placed on an abstract method, the constraint is applied to the implementations of that method, using the inheritance rules defined in Section 4.1. Figure 4.9 illustrates a precondition defined on an abstract method.

```

public abstract C1
{
    OCL
    [
        "context C1::Square(i : Integer) : Integer"
        "pre: i > 0"
    ]
    public abstract int Square(int i);
}

```

Figure 4.9: *Precondition applied to an abstract method*

4.11 Constraints and The *new* Modifier

If the *new* modifier is used on a new member to hide the inherited member, then any OCL preconditions and postconditions from the inherited member are also hidden.

If the superclass contains an invariant that is used in the context of the new member, and the invariant type does not match the new member's type, a compile-time error will be generated. The error is generated because the invariant of the superclass is referencing a member that is of a different type than the hidden member. Figure 4.10 illustrates a case when the error will be generated. In Figure 4.10, the invariant states that the double value cannot be equal to infinity. The infinity check is performed via the “isInfinity” method defined on double types. When the “Real” class is used as a superclass for the “Integer” class, the invariant is propagated to the “Integer” class. However, in the “Integer” class, the “_value” attribute is being represented by an int type instead of a double type, and the int type does not define the “isInfinity” method. That is, the compiler will generate a compile-time error that will indicate that the “isInfinity” method is not defined on the int type.

```

OCL
[
  "context Real"
  "inv: not _value.isInfinity()"
]
class Real
{
  protected double _value = 0.0;
}
class Integer : Real
{
  protected new int _value = 0;
}

```

Figure 4.10: Error case caused by the use of OCL and the new modifier

4.12 Constraints and the *override* Modifier

The C# *override* modifier is the opposite of the *new* modifier. The *override* modifier has the same semantics as inheritance in C++. The *override* modifier explicitly states that the default inheritance rules are to be used. For example, in Figure 4.11 the precondition specified on “Dimensions::Area()”, would be applied to “Circle::Area()” using Findler’s method[8].

```

public class Dimensions
{
    public const double pi = Math.PI;
    protected double x, y;
    public Dimensions (double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    OCL
    [
        "context Dimensions::Area() : Real"
        "pre: x > 0"
    ]
    public virtual double Area()
    {
        return x*y;
    }
}
public class Circle: Dimensions
{
    public Circle(double r): base(r, 0)
    {
    }
    public override double Area()
    {
        return pi * x * x;
    }
}

```

Figure 4.11: *Use of the OCL with the override modifier*

4.13 Summary

Preconditions and postconditions can be placed on methods, constructors, destructors, and the accessors of properties, indexers, and events. Preconditions are checked before the body is allowed to execute, and postconditions are checked after the body has executed. Class invariants can be placed on classes and structures, but not on interfaces, as they do not contain attributes.

Delegates allow constraints to be placed on a family of operations. When a delegate invokes one or more methods, the caller can be sure that every method assigned to the delegate will fulfill the constraints specified at the delegate level.

Inheritance of constraints is performed via the method outlined by Robert Findler's PhD thesis[8]. Findler's method updates Meyer's method by allowing blame to be assigned to the class hierarchy if a subclass is incorrectly using preconditions or postconditions.

With the definition of what it means to assign a contract to a given C# program element complete, we will now look at the process of converting OCL expressions into actual code.

Chapter 5

Resolution

This chapter will examine the compilation process of transforming the OCL expressions into executable code. The chapter will begin with a look at the modifications required to the stock Mono compiler that will allow for the compilation of OCL expressions. Next, an explanation of how the OCL type library is implemented in C# is presented. With the OCL type library implemented, we then look at how each of the various types of OCL expressions are resolved.

5.1 Mono Compiler Preparation

The Mono compiler is an open source C# compiler provided by the Ximian company[47]. It consists of forty-two files that provide over forty-six thousand lines of code. The compiler is object-oriented in that each C# element, both structural and behavioral, is represented by a class. For example, the notion of a C# *class* is represented by a class called “Class”. As well, the concept of an *if* statement is represented by a class called “If”.

The Mono compiler has a number of phases. The following sections will look at each of these phases in turn and discuss the modifications made to support the OCL.

5.1.1 Lexical Analyzer

The first phase consists of the lexical analyzer. The lexical analyzer takes as input the various C# source files that make up a given project. The lexical analyzer then takes the input and translates it into tokens that will be used in phase two.

In terms of the OCL, the lexical analyzer was modified to support the new *OCL* keyword discussed in Chapter 3. As previously discussed the OCL expressions are contained within C# style literal strings, so that the lexical analyzer will treat the entire OCL expression as a string literal token.

5.1.2 The Parser

Once the lexical analyzer has converted the source files into tokens, the tokens are passed to the parser. The parser is implemented using JAY[47]. JAY is a C# port of Yacc developed at Berkeley. The parser takes the tokens as input. It provides basic syntax checking and constructs a parse tree from the input tokens.

As previously stated, the Mono compiler contains a class for each C# language element. The parser creates instances of these classes as needed, and uses these element classes to construct the parse tree.

The modifications to the C# parser and the introduction of the OCL parser have already been discussed. The C# grammar that is used by JAY to build the parse tree has been modified, as discussed in Chapter 3. In addition to the grammar modifications, each of the structural C# element classes that can have OCL expressions assigned to them have been modified to store the OCL parse trees that were created by the OCL parser defined in Chapter 3. The OCL parse trees are assigned to the C# element that was specified in the OCL context expression. For example, the “Method” class has been augmented to have three lists of OCL parse trees. One list stores the preconditions, while the other two store the postconditions and body rules respectively.

By having the OCL parse trees assigned to the correct C# element class, the resolution of the OCL expressions can be done in the context of their owner.

5.1.3 Parent Class Resolution

Once the C# parse tree has been constructed, the Mono compiler begins the first of two resolution passes. The first resolution pass resolves the parents for interface, class, and structure definitions. The first resolution pass also constructs the type hierarchy. The second resolution pass performs the actual semantic analysis. It will be discussed in Section 5.1.5.

No major modification to the first resolution pass is required to implement integration of the OCL. The only minor modification is that the types defined in the OCL type library are added to the C# type hierarchy. These types include the OCL primitive data types, such as “Integer” and “String”, as well as more complex types such as “Tuple”, and “Set”. The details of the OCL type library will be discussed in Section 5.2.

5.1.4 OCL Semantic Analysis

Upon successful completion of the parent class resolution phase, the Mono C# compiler is paused. At this point, each of the OCL expressions is resolved using the C# parse tree and the type hierarchy. The details of the resolution process will be discussed in Section 5.3. Once an OCL expression is resolved, the result of the resolution process is one or more C# statements. These C# statements are inserted into the existing C# parse tree.

5.1.5 C# Semantic Analysis

Once the C# parse tree has been augmented with additional C# statements that represent the various OCL expressions, the Mono compiler is restarted. The Mono compiler completes the resolution of the C# parse tree.

5.1.6 Code Generation

Upon the successful completion of both OCL and C# semantic analysis, the C# parse tree contains enough information to generate executable code. The code generation is accomplished by traversing

the parse tree and generating the corresponding code for each element via the “System.Reflection.Emit” API. The “System.Reflection.Emit” API provides a set of classes that can be used to create new managed modules and assemblies. By using this API, the compiler is saved from having to perform the low-level code generation activities. The code generation is performed via the object-oriented framework provided by the API[26].

5.1.7 Summary

The main design decision, with respect to compiler preparation, was deciding where to stop the C# compilation process so that the OCL expressions could be integrated. The compiler was stopped mid-way through the semantic pass. By stopping the compiler during the semantic pass, we are then able to graft each of the OCL expressions onto the main C# parse tree. The existing C# parse tree can also be used to represent a virtual model for performing the semantic analysis of the OCL expressions. Once each of the OCL expressions is inserted into the main parse tree, the C# compilation process is resumed. The remainder of the compilation process generates the byte code for the C# parse tree. By inserting C# code to represent the OCL expressions into the main parse tree before the code generation process, the need to write a specialized code generator is eliminated.

With the modifications to the Mono C# compiler complete, the OCL specific implementation details of the integration will be examined.

5.2 The OCL Type Library Integration

The OCL specification contains a rich type library[32]. Before the OCL expressions can be resolved, the type library must be implemented in C#. In order to keep the compiler modular, a separate C# library has been created to represent the OCL type library.

The OCL type library contains definitions for various OCL types. The most important types consist of the four OCL primitive types: “Boolean”, “Real”, “Integer”, and “String”. In addition to the primitive

types, the OCL type library also defines various collection types: “Bag”, “Set”, “OrderedSet”, and “Sequence”. Each of the collection types are derived from an abstract collection type: “Collection”. In addition to defining the various types, the OCL type library also contains a complete list of the operations permitted on each type. Figure 5.1 illustrates the contents of the OCL type library.

Figure 5.1: The OCL type library in a UML class diagram

As Figure 5.1 illustrates, each of the types defined in the specification of the OCL type library correspond to the names provided in the specification[32]. There are two exceptions to this: the “Boolean” and “String” types are prefixed with “Ocl” to avoid ambiguity with the C# “Boolean” and “String” types.

In order to implement the OCL type library in C#, a mapping between each OCL type to a corresponding C# type as been defined. Table 5.1 outlines the mappings between the two languages. The following sections will look at each of the types defined in the OCL type library.

OCL Type	C# Type	C# Class Name
OclAny	Object	OclAny
OclVoid	Null	OclVoid
Tuple	<None>	Tuple
Boolean	Bool	OclBoolean
Real	Double	Real
Integer	Int32	Integer
String	String	OclString
Collection	ArrayList	Collection
Bag	ArrayList	Bag
Set	ArrayList	Set
OrderedSet	ArrayList	OrderedSet
Sequence	ArrayList	Sequence
OclType	Type	OclType
OclModelElement	Object	OclModelElement
OclState	Object	OclState

Table 5.1: Mapping between the OCL type library and C# types

5.2.1 Base Types

The “BaseTypes” package illustrated in Figure 5.1 contains the “OclAny”, “OclVoid”, and “Tuple” classes. These types provide the base for all types, in not only the type library, but also all types defined under the OCL. We will now look at each of the base types in detail.

OclAny

The “OclAny” type is the supertype of all types defined in the OCL model, as well as the primitive types defined in the type library[32]. The collection types defined in the type library are the only exception, and are not subtypes of “OclAny”. The “OclAny” type is analogous to the “Object” type defined in C#.

To this effect, the “OclAny” class encapsulates an instance of “System.Object”. The operations defined on the “OclAny” class are performed on the underlying “System.Object” instance. The majority of the operations defined on the “OclAny” type are used for type checking and type conversions. All the operations defined on the “OclAny” type are implemented with the exception of two: “oclIsNew”, and “allInstances”.

The “oclIsNew” operation is used in a postcondition to determine if the object that is encapsulated by the “OclAny” type was created during the execution of the method’s body. In other words, did the object instance not exist during precondition time? The “oclIsNew” operation is implemented by the compiler using two *if* constructs. The first is placed in the precondition to verify that the object is equal to null. The second *if* construct is placed in the postcondition to ensure that the object now contains a non-null value; the object has been created. If one or both of the *if* constructs fail, then the postcondition containing the “oclIsNew” operation will fail.

The “allInstances” operation returns all the active instances of the corresponding type. For example, if the “allInstances” operation is called on an instance of the “Person” type, all of the instances of type “Person” that are active in the system will be returned as a set. For technical and design reasons, the “allInstances” operation is not implemented. The reasons will be discussed in Chapter 7.

As C# already has a default type, and to avoid having to modify every class defined in the C# model, the types defined outside the type library are not actually derived from the “OclAny” class. In order to allow the operations defined on the “OclAny” type to be executed on objects that do not inherit from the “OclAny” type, the compiler will fake the inheritance, and instead of calling the requested “OclAny” method, the compiler will simply replace the method call with the actual code. The technical details of implementing the “OclAny” type will be discussed later in the Chapter.

OclVoid

The “OclVoid” type that conforms to all other types[32]. The “OclVoid” type has a single instance called “OclUndefined”. Any method called on the “OclUndefined” instance results in the same “OclUndefined” instance. There are three exceptions to this rule. A boolean value of true “or”ed with “OclUndefined” will result in a true value. Similarly, a boolean value of false “and”ed with “OclUndefined” will result in a false value. The final exception is that a boolean value of false implied with “OclUndefined” will result in a true value.

The “OclVoid” type only defines a single operation: “oclIsUndefined”, which always returns true.

The compiler maps the “OclVoid” type to the C# *null* keyword. There is one exception to this mapping. In the case where ”OclVoid” is used in context expression, the “OclVoid” type is mapped to the C# “void” type. See Figure 5.2 for an example.

```
class MyClass
{
    OCL
    [
        "context MyClass::Method1(p : Person) : OclVoid"
        "pre: not p.oclIsUndefined()"
    ]
    Public void Method1(Person p)
    {
        . . .
    }
}
```

Figure 5.2: Example use of the OclVoid type

The context line in Figure 5.2 will map the “OclVoid” type to the C# “void” type, to match the precondition with “Method1”. The precondition will raise an exception if “p” is equal to null.

Most of the mapping of the “OclVoid” type to C# is performed via the compiler, and the actual class is not used. The class is only used as storage for the single “OclUndefined” instance. The instance is used in logical comparisons.

Tuple

The tuple is not formally defined in the OCL type library. A tuple consists of named parts, each of which can have a distinct type[32]. As the OCL is a strongly typed language, each tuple has its own type. Each tuple type does not have a name. The tuple type is defined by the value of the tuple. Figure 5.3 illustrates some tuple values.

```
Tuple {name: String = 'Dave', age: Integer = 25}
Tuple {a: Collection(Integer) = Set{1, 2, 4},
      b: String = 'foo',
      c: String = 'bar' }
```

Figure 5.3: Tuple values

As Figure 5.3 shows, the parts of a tuple are enclosed by braces, and separated by commas[45]. The type names are optional, and the order of the tuple parts is unimportant. Figure 5.4 shows three tuples that all have the same value.

```

Tuple {name: String = 'Dave', age: Integer = 25}
Tuple {name = 'Dave', age = 25}
Tuple {age = 25, name = 'Dave'}

```

Figure 5.4: *Tuples all with the same value*

Figure 5.3 and Figure 5.4 illustrate the use of a tuple value. Each tuple value defines a tuple type. A tuple type consists of the names and types of the parts that make up a tuple. Two tuple types are considered equal if each name and corresponding type of one tuple type exists in the other tuple type. The tuple types must also have the same number of tuple parts. Figure 5.5 illustrates an example of a tuple type declaration.

```

TupleType(name: String, age: Integer)

```

Figure 5.5: *Tuple type declaration*

As Figure 5.5 illustrates, the parts of the tuple are enclosed in parenthesis, and separated by commas. The type names are required, but the order of the parts is not important[45].

The parts of an instance of a tuple type can be accessed by using the names contained within the type. Figure 5.6 illustrates an example of accessing the elements of a tuple value.

```

Tuple {name: String = 'Dave', age: Integer = 25}.age
Tuple {name: String = 'Dave', age: Integer = 25}.name

```

Figure 5.6: *Tuple element access*

The tuple type does not directly map into an existing C# type. In our work, an instance of a tuple is represented by two hash tables. The first hash table contains a map between the tuple names and the actual values. The second hash table contains a map between the tuple names and their corresponding types. As tuples are strongly typed, there may be several different tuple types defined in a system. The compiler will handle the definition and usage of the various tuple types, to ensure the correct type relationships. The actual tuple storage occurs in the C# “Tuple” class, which is a generic tuple storage class.

With the three OCL base types defined, we will now examine the OCL’s four primitive types: “Boolean”, “Real”, “Integer” and “String”.

5.2.2 Primitive Types

The OCL defines only four primitive types. Each of the primitive types is represented by a C# class. Each class contains a C# primitive data type that stores the actual value, and a method for each operation defined on the type.

Each of the C# classes that represent the OCL primitive types also contain implicit conversion operators. The implicit conversion operators allow for the classes that represent the OCL types to be used where their corresponding C# primitive type can be used. The implicit conversion operators allow the classes to be used directly in expressions with the C# primitive types, as well as with literal expressions.

Boolean

A value of the “Boolean” type can be one of two values: true or false. The “Boolean” type in OCL is mapped to the C# “bool” primitive. The OCL “Boolean” type defines all the familiar boolean operations such as: or, and, exclusive or (xor), negation (not), equality (=), and non-equality (<>). The uncommon operation, implies, is also defined on the OCL “Boolean” type. The implies operation states that the result of the expression is true if the first and second operands are true. If the first operand is false, the entire implies expression is true, regardless of the second operand. Figure 5.7 shows some example boolean expressions.

```
not true
age() > 21 and age() < 65
age() <= 12 xor cards->size() > 3
```

Figure 5.7: *OCL boolean expressions*

Real

The OCL “Real” type represents the mathematical concept of real values[45]. C# contains various primitives for representing a real number. The implementation uses the “double” primitive type to represent an OCL real number. The “Real” type defines all common mathematical operations that involve real numbers. For a complete list, please see the OCL specification[32].

Integer

The OCL “Integer” type represents the mathematical natural numbers[45]. As in mathematics, “Integer” is a subtype of “Real”. The implementation uses the “int” primitive type to represent an OCL integer. Like the “Real” type, the “Integer” type defines all the common mathematical operations that involve natural numbers. For a complete list, please see the OCL specification[32]. Figure 5.8 demonstrates the use of “Integer” and “Real” types[45]. Each of the expressions in Figure 5.8, are of the “Boolean” type, and all result in true.

```
2654 * 4.3 + 101 = 11513.2
(3.2).floor() / 3 = 1
1.175 * (-8.9).abs() 10 = 0.4575
12 > 22.7 = false
12.max(33) = 33
33.max(12) = 33
12.mod(2) = 1
13.div(2) = 6
33.7.min(12) = 12.0
-24.abs() = 24
(-2.4).floor() = -3
```

Figure 5.8: OCL integer and real expressions

String

The OCL “String” type represents a sequence of characters. Literal strings in the OCL are denoted by enclosing single quotes, such as ‘Dave’. The OCL “String” type supports the following operations: “toUpper”, “toLower”, “size”, “substring” and “concat”. Figure 5.9 illustrates the usage of the OCL “String” type. Each of the expressions in Figure 5.9 is of the “Boolean” type, and results in true.

```
‘Dave’.size() = 4
(‘Dave’ = ‘Toby’) = false
‘Dave’.concat(‘ and Toby’) = ‘Dave and Toby’
‘Dave’.toUpper() = ‘DAVE’
‘Dave’.toLower() = ‘dave’
‘Dave and Toby’.substring(6, 13) = ‘and Toby’
```

Figure 5.9: OCL string expressions

In our implementation, the OCL “String” type is represented by the C# “string” primitive. Each of the operations defined on the OCL “String” type are implemented by a method, which operates on

the underlying C# primitive. For more information of the OCL “String” type, please see the OCL specification[32].

The OCL “String” type completes the integration of the OCL primitive types into C#. The integration will continue by looking at the model element types.

5.2.3 Model Element Types

The OCL type library defines three types that allow the modeler to refer to elements defined in the model[32]. These three types are “OclModelElement”, “OclState”, and “OclType”. The next sections will look at each of these three types in turn.

OclModelElement

The “OclModelElement” type is an enumeration. For each element defined in the model there is a corresponding enumeration literal. For example, if a developer defines a class called “Person”, there will be an element in the “OclModelElement” enumeration called “Person”. As the “OclModelElement” type is not really a type but an enumeration, it does not have any operations defined on it. “OclModelElement” allows for access to the various elements defined in the model.

The integration of the OCL and C# occurs at the code level. Thus, the compiler allows for any OCL expression to access elements defined in C# directly. The use of “OclModelElement” is implied and implemented by the compiler. The “OclModelElement” type does not need to be explicitly implemented. In terms of the implementation, two “OclModelElements” are considered equal if the underlying C# objects are equal.

OclState

The “OclState” type is also an enumeration. For each state in the model there is a corresponding enumeration literal[32]. As the C# language does not directly define the notion of a state, an instance of the

“OclState” type is defined by an instance of an object. The value of an instance of the “OclState” type is defined by the value of an object.

In the implementation, the state of an object is defined as the value of all of the object’s attributes. Two objects are in the same state if every attribute defined on the two object’s contains the same value, and if the objects are instances of the same type. Like the “OclModelElement” type, the “OclState” type is implemented implicitly by the compiler. The actual “OclState” type is not explicitly implemented. If the OCL was only used at the code level, the “OclModelElement” and “OclState” types would be removed entirely from the system. As it is possible that the OCL used at the code level came directly from a model at a higher level of abstraction, the use of these types is supported at the code level. If the “OclModelElement” or “OclState” syntax is encountered, the compiler will transform the statement to directly access the corresponding element, rather than use the enumerations.

OclType

The third and final model element type is the “OclType” type. The “OclType” type is used to represent each type defined in the model. In the terms of C#, there would be an “OclType” type for every type defined under C#.

C# already provides for the type functionality provided by the “OclType” type in the “Type” class. The “Type” class is a general-purpose class for getting type information about objects in a system. The “Type” class can be used to define new types dynamically. The implementation of “OclType” simply encapsulates an instance of the C# “Type” class. The “OclType” does not define any specialized operations. The only non-standard operation defined on the “OclType” type is an implicit conversion operation. The implicit conversion operator allows an “OclType” object to be used in a context where a C# “Type” object is expected, and vice versa.

Model Elements Summary

As illustrated by the previous sections, the model element types are used for referencing elements in the system model. At any level of abstraction above the actual source code, these model element types play

an important role in allowing OCL expressions to reference elements, states, and types that are defined using a modeling language, such as the UML. In the case of our integration, at the source code level the reference model is represented by a parse tree. The parse tree already contains all of the element, state and type information. The model element types do not need to be explicitly defined, as the OCL compiler already has implicit access to the elements contained within the parse tree. The implementation of the OCL type library contains wrapper classes that represent the three model element types. The “OclModelElement” and “OclState” classes can encapsulate any object instance. The “OclType” class encapsulates the C# “Type” class.

5.2.4 Collections

As the use of collections in systems is common, the OCL type library contains a series of collection classes. The OCL type library defines four concrete collection types, and one abstract type. The abstract collection type, “Collection”, is used to define various operations that are common to all collection types. The four concrete collection types: “Set”, “OrderdSet”, “Bag”, and “Sequence”, can be used directly in expressions. Each of the collection types is actually a template type with a single parameter[32].

A concrete collection type is created by specifying a type for the template parameter. For example, “Set(Integer)” is a concrete collection type. As the OCL is a strongly typed language, all collection types must be explicitly defined, and a collection may not contain elements of different types. The following sections will examine each of the five OCL collection types in detail.

Collection

The “Collection” type is the abstract supertype for all of the collection types defined in the OCL type library[32]. A collection can contain zero or more elements. If an element occurs twice in the collection, there are two elements in the collection. As previously, stated the “Collection” type defines various operations that are common to all of the collection types: “count(object)”, “excludes(object)”, “excludesAll(collection)”, “includes(object)”, “includesAll(collection)”, “isEmpty()”, “notEmpty()”, “size()”, and “sum()”.

The “count” operation determines the number of times that the given object appears in the collection. If the object does not appear in the collection, a zero value is returned. The “excludes” and “excludesAll” operations yield a result of true if the given object is not contained within the collection. The “exclude-sAll” operation takes a collection of elements and will return true only if each of the elements are not in the collection. The “includes” and “includesAll” operations are the inverse of the excludes operations; they return true if the object is contained within the collection. In the case of “includesAll”, every element in the parameter collection must exist in the source collection for a true result. The “isEmpty” and “notEmpty” operations determine if the collection does not contain any elements, or if there is one or more element in the collection. The “size” operation returns the number of elements in the collection. Finally, the “sum” operation returns the value of adding each element of the collection together. In order to use the “sum” operation, the element type of the collection must support the use of the addition (+) operator. In the case of the predefined OCL types, the “sum” operation can be used on values of the “Real” and “Integer” types.

A collection may be defined via a collection constant or a collection type. A collection constant represents a literal representation of a collection. A collection type, defines a new collection using a specified element type. Figure 5.10 illustrates an example of both collection types. Figure 5.10 defines a method that returns a set whose elements are of the type “Integer”. The body of the method is defined to be a literal set of values.

```

OCL
[
  "context MyClass::Method1() : Set(Integer)"
  "body: Set { 1, 2, 3, 4, 34, 23 }"
]

```

Figure 5.10: *Use of collection types*

In terms of implementation, the OCL “Collection” is defined in terms of a dedicated C# class, “Collection”. The C# “Collection” class encapsulates an “ArrayList” that represents the underlying collection. As the “ArrayList” can be used to represent any type of object in the system, the compiler will enforce the use of the correct element type. The use of a collection with incorrect element types results in a compile-time error. The compiler also enforces the typing of strong collections. For example, a collection of “Integer” types cannot be used or compared with a collection of “Boolean” types.

For ease of integration with C#, the “Collection” class also contains implicit conversion operators. The implicit conversion operators allow any OCL collection class to be used as an “ArrayList” and any “ArrayList” to be used as an OCL collection. When an “ArrayList” is converted to the requested OCL collection type, the elements of the collection are checked, at runtime, to ensure that they do not violate any of the OCL collection rules. These collection rules depend on the type of collection being used. In terms of the generic “Collection” class, the only rule is that each of the elements contained within the collection must be of the same type.

The implementation uses the following rules to determine if two elements of a collection are equal. If a collection element is a value type, such as an integer, the actual integer value will be used to determine the element’s value. If the collection element is a reference type, the “Object.Equals” method will be used to determine the elements value. If the “Object.Equals” method does not exist, the default implementation will be used. The default implementation uses the value of the reference itself. If two references are used to reference the same object, they will not be deemed equal unless the object provides an implementation for the “Object.Equals” method[26]. The four concrete collection types will now be examined.

Set

The OCL “Set” collection is defined to be a mathematical set[32]. The set contains elements without duplicates. The elements of a set are not ordered. If an element is added to a set that already contains the element, an exception is raised.

There are multitudes of operations defined on the “Set” type[32, 45]. The operations permit the addition, retrieval, and removal of elements from a set. Some of the operations defined on the “Set” type are illustrated in Figure 5.11.

In C#, the “Set” type is implemented via the “Set” class. The “Set” class is subclassed from the “Collection” class outlined in the previous section. The “Set” class implements the operations that are set specific, as well as strengthens the rules for conversion from an “ArrayList” to a “Set”. The additional

```

Set { Set {1, 2}, Set {2, 3}, Set {4, 5, 6} }.flatten() =
    Set {1, 2, 3, 4, 5, 6}
Set {1, 4, 7, 10} - Set {4, 7} = Set {1, 10}
Set {1, 4, 7, 10}.symmetricDifference(Set {4, 5, 7} ) =
    Set {1, 5, 10}
Set {1, 2, 4}.including(3) = Set {1, 2, 3, 4}

```

Figure 5.11: *Some operations defined on the Set type*

rule for the “Set” type is that if the same element exists more than once in the “ArrayList”, the “ArrayList” cannot be converted into a set, and an exception is raised.

OrderedSet

The OCL “OrderedSet” collection type is defined as a set whose elements are ordered[32]. Like the “Set”, the “OrderedSet” type does not contain duplicate elements. If an element is added to an ordered set that already contains the element, an exception is raised.

Each of the operations that are defined on the “Set” type, are also defined on the “OrderedSet” type. The “OrderedSet” type also contains eight operations that involve the ordering of the elements within the ordered set. A summary of these operations is as follows:

- first() and last(): Returns the first and last element of the set.
- at(x : Integer): Returns the element at the location specified by x. The indices are one based.
- indexOf(x : OclAny): Returns an integer indicating the position of the given element. An index of zero indicates that the given element does not exist in the set.
- insertAt(x : Integer, y : OclAny): Inserts the object, y, in the set at the position specified by x. The result of the operation is a new ordered set with the item inserted. The original set is unchanged.
- subOrderedSet(x : Integer, y : Integer): Returns a new ordered set whose elements are the elements of the original ordered set beginning with the element located at index x, up to and including the element located at index y. The order of the elements is unchanged.
- append(x : OclAny) and prepend(x : OclAny): Returns a new ordered set with the new element added to end or beginning of the existing ordered set, respectively.

```

OrderedSet { 'a', 'b', 'c', 'd', 'e' }.first() = 'a'
OrderedSet { 'a', 'b', 'c', 'd' }.last() = 'd'
OrderedSet { 'a', 'b', 'c', 'd', 'e' }.at(3) = 'c'
OrderedSet { 'a', 'b', 'c', 'd', 'e' }.indexOf('c') = 3
OrderedSet { 'a', 'b', 'c', 'd' }.insertAt(3, 'X') =
    OrderedSet { 'a', 'b', 'X', 'c', 'd' }
OrderedSet { 'a', 'b', 'c', 'd', 'e' }.subOrderedSet(2, 3) =
    OrderedSet { 'b', 'c' }
OrderedSet { 'a', 'b', 'c', 'd' }.append('X') =
    OrderedSet { 'a', 'b', 'c', 'd', 'X' }

```

Figure 5.12: *Some operations defined on the OrderedSet type*

Figure 5.12 illustrates the usage of some of the operations outlined above.

As in the other collection classes, the “OrderedSet” type is implemented in its own C# class, “OrderedSet”. The “OrderedSet” class extends the “Set” class, and adds the additional functionality outlined previously. The only note regarding the implementation of the “OrderedSet” class is that when converting from an “ArrayList” to an “OrderedSet” (via the implicit conversion operator), the order used to create the “OrderedSet” is the order that the elements are stored in the “ArrayList”. Depending on how the “ArrayList” was constructed, this order may be random. If, however, an “OrderedSet” is converted to an “ArrayList”, and then back to an “OrderedSet”, the order of the elements will be preserved as long as the “ArrayList” is not modified.

Bag

The OCL “Bag” type is a collection which allows duplicate elements[32]. In terms of operations defined on the “Bag” type, the “Bag” type defines similar operations as the “Set” type. The only exceptions are that some of the operations, which under the “Set” type expected parameters of type “Set”, now expect parameters of type “Bag”[32, 45].

In terms of implementation, the “Bag” type is represented by the C# class “Bag”. The “Bag” class is similar to the “Set” class, and will not be discussed any further.

Sequence

The OCL “Sequence” type is a collection where the elements are ordered[32]. Unlike the “OrderedSet” type, a sequence may contain duplicate elements. The “Sequence” type defines the same operations that were presented for the “OrderedSet” type. The only exception is that the “Sequence” type does not contain the “subOrderedSet” operation, but rather contains a “subSequence” operation. The “subSequence” operation takes the same parameters and performs the same function as the “subOrderedSet” operation, except that it results in a new “Sequence” as opposed to a new “OrderedSet”[32, 45].

The implementation of the “Sequence” type is handled by the “Sequence” class. The “Sequence” class is similar to the “OrderedSet” class, with the exception of the operations previously discussed.

5.2.5 Iterators

Each of the five collection types defined in the OCL type library define a set of predefined iterators for performing operations on each element of a collection. The iterators cannot be implemented via a static type library, as the iterators rely heavily on the element type. In order to accommodate the iterators, they are implemented via compile-time code as encountered in the OCL expressions. Code for an iterator will only be generated if the iterator is used. The details of the code generated for an iterator are discussed in Section 5.3.2.

5.2.6 Instances as Collections

As the collection classes defined in the OCL type library contain powerful operations, it is possible to use a single instance as a collection. In the OCL expressions examined up to now, a method call on an object is performed by using the “.” operator. Figure 5.13 illustrates such a method call.

```
Set {1, 2, 4}.size() = 3
```

Figure 5.13: *OCL method call*

In Figure 5.13, the “size” method is being called on the literal set object. In order to use a single instance as a collection the “->” operator can be used. In Figure 5.14, the “->” operator is used so that the single

instance of the “Person” class is treated as a collection. The “->” operator creates a new instance of the “Set” type, and uses the left side of the expression to be the single element of the “Set”. Returning to Figure 5.14, the expression creates a new set with the instance of the “Person” class as the lone element. The “size” operation is then executed on the new set.

```
aPerson->size() = 1
```

Figure 5.14: *OCL method call on a instance used as a collection*

The ability to use instances as collections is implemented by the compiler. The compiler will insert code in to the parse tree, which will create a new collection and then place the single element into the collection. The details of this process are discussed in Section 5.3.2.

5.2.7 Non-OCL Specific Elements

The elements of the OCL type library discussed up to this point represent the complete OCL type library as defined in the OCL specification[32]. In order to support the OCL within C#, a few additional items were added to the OCL type library.

Pair

In the context of the collection classes, a “Pair” class is defined. The “Pair” class maintains two objects. The first object, which must implement the “Comparable” interface, acts as a key. The second object, on which there is no restriction, acts as a value. In short, the “Pair” class represents a key value pair. This key value pair is used during the evaluation of some of the collection operations, as well in some of the iterator implementations.

Query Attribute

In the OCL, all methods called in the context of an OCL expression must be side effect free. This corresponds in C# to a query method. A query method is an operation that does not modify the state of the system.

When the resolution of an OCL expression involves calling a method defined on an object, the compiler must ensure that the method is in fact a query method. This is a non-trivial task. Unlike C++, C# does not define the *const* keyword. In order to mark a method as a query method, the “Query” attribute is used. Figure 5.15 illustrates the usage of the “Query” attribute. If a method is called by an OCL expression, and the method is not marked with the “Query” attribute, a compile-time error is generated. The “Query” attribute is automatically assigned to any method that is defined using an OCL expression, as all OCL expressions are guaranteed to be side effect free by definition.

```
[Query]
int GetNumber()
{
    return 1;
}
```

Figure 5.15: *Usage of the Query attribute*

The reader should note that the “Query” attribute is used by a developer to indicate that a method does not contain any side effects. The onus is on the developer to ensure that the method in fact does not contain any side effects. The compiler will not enforce the correct use of the “Query” attribute. The actual implementation of the “Query” attribute class is located in the OCL type library.

OCL Internal Exception

The “OCLInternalException” class is a specialization of the C# “Exception” class. An “OCLInternalException” is thrown when there is an unhandled or unexpected case in the implementation of the OCL type library. The “OCLInternalException” only contains a message indicating the nature of the exception. The exception is only raised when there is an error in the OCL type library. The exception does not mean that an OCL constraint was violated, but rather that there was an error (bug) in the implementation of the OCL type library. The “OCLInternalException” is never caught by the implementation.

5.2.8 Handling Constraint Violation

When an OCL constraint is violated at runtime, the system will execute one of three methods for handling the constraint failure. The first method is to throw an OCL exception. The second method is to

display a dialog box explaining the violation. Once the user has dismissed the dialog box, the application will be terminated. The third, and final, method is a combination of methods one and two. The dialog box is displayed followed by the OCL exception. The method used to handle the failure of OCL constraints is specified via a compiler option. The actual exception and dialog box classes are defined as part of the implementation of the OCL type library. The next two sections will look at each in turn.

OCL Exception

The “OCLException” class is a specialization of the C# “Exception” class. An instance of the “OCLException” class contains four elements. The first element indicates the type of constraint that failed. Valid constraint types consist of preconditions, postconditions, and class invariants. The second element is the context or location where the constraint failed. The context is a string representation of the context expression that was used in the OCL expression. The third element is a string that indicates the C# structural element that is at fault. Blame is allocated in accordance with Findler’s method[8]. The fourth, and final element is a string that stores the OCL expression as typed by the user.

The “OCLException” also contains a “ToString” method that generates a single string that represents the exception. The resultant string contains all of the information contained within the four elements, plus a complete stack trace. Figure 5.16 shows an example exception. The figure illustrates that the postcondition located on the “GetName” method, which is defined in the “NavTests” class, has failed.

```
OCL Exception: Constraint Failed
Expression Type: Post-Condition
Context: 'DaveArnold::OCLTesting::NavTests::GetName'
OCL Expression:
    context NavTests::GetName(px : PersonData) : String
    post: result = px.px.name
Fault: Supplier
Stack Trace:
    at DaveArnold.OCLTesting.NavTests.GetName(PersonData px)
    at DaveArnold.OCLTesting.NavTests.RunTests()
    at DaveArnold.OCLTesting.OCLTestDriver.Main()
```

Figure 5.16: *Postcondition failure via an OCL exception*

OCL Constraint Failure Dialog

The “OCLConstraintFailureDialog” is a specialization of the “Windows.Forms.Form” class. The “OCLConstraintFailureDialog” class contains the same elements as the “OCLException” does. The only difference is that the information is displayed in the context of the dialog box. A colour image of the dialog box is shown in Appendix B.

5.2.9 Summary

The OCL defines a rich type library. The type library contains definitions for the underlying core types, the four OCL primitive types, the model element types, and the collection types. The underlying core types specify a group of operations that can be applied to any object. The four primitive OCL types consist of “Boolean”, “Real”, “Integer”, and “String” types. The model elements allow for access to the underlying model. The collections consist of “Set”, “OrderedSet”, “Bag”, and “Sequence” types.

Some additional types were added to the implementation of the OCL type library in order to support the integration with C#. These types include classes to support query methods, and the raising of exceptions.

In terms of implementation, a separate C# class has been created for each OCL type defined in the type library. Instead of creating dedicated C# classes, static methods could be defined to implement the operations defined on each OCL type. The use of static methods would remove the need for constant type conversions between the native C# type and the corresponding OCL type. C# allows for implicit operator definitions. These implicit operators allow for automatic type conversion between our type library classes and the underlying C# types. For this reason, and to improve modularity, the class model has been chosen over the static method model. For the integration with languages that do not support the use of implicit operators, the use of the static method model would be recommended.

With the mapping of the OCL type library complete, we can now move on to the next step: resolve and map each of the OCL expressions into one or more C# statements. The next section will look at each of the OCL expressions and explain what level of syntactic checking is performed, as well as what resultant C# code is produced.

5.3 Resolution of OCL Expressions

In Chapter 3, the OCL grammar was examined. The OCL grammar is fed into a parser generator that creates the actual OCL parser. The OCL parser produces an abstract syntax tree (AST) for each OCL expression. The next step is to resolve each of the ASTs into one or more C# statements.

The OCL ASTs are made up of classes. Each class represents a different type of OCL expression. Figure 3.6 in Chapter 3 provides a UML diagram of the classes. The following sections will examine each of type of OCL expression and the resolution process.

In the context of resolution, there are two types of expressions: expressions that can be resolved into a constant value and expressions that cannot be resolved into a constant value. When an expression needs to be resolved, a constant resolution is attempted first. The constant value is used in C# if a constant resolution can be achieved. If the expression cannot be resolved into a constant, then the expression is resolved into one or more C# statements. In order to perform the resolutions, the C# AST is used as a reference model for type lookups.

5.3.1 Constant OCL Expression Resolution

Four categories of OCL expressions can be resolved into a C# constant: literal expressions, constant variables, some methods, and a select group of attribute call expressions.

Literal Expressions

The simplest expressions to resolve are literal expressions. Literal expressions are a value that is placed directly in the code, rather than an expression to calculate the value. There are eight types of literal expressions: the four primitive types, enumerations, literal tuples, literal collections, and void literals.

The four primitive types are simply mapped to their respective C# primitive type. The OCL “Boolean” literal is mapped to a C# “bool” literal; the OCL “Integer” literal is mapped to a C# “int” literal; the

OCL “Real” literal is mapped to a C# “double” literal; and the OCL “String” literal is mapped to a C# “string” literal.

An OCL enumeration has the form of “EnumerationType::Value”. The first step in resolving the enumeration is to locate the enumeration type in the C# AST. If the enumeration type cannot be located, a compile-time error is generated and the resolution fails. Once the enumeration type is located, the enumeration type is checked to see if the requested value exists. If the enumeration type exists, but the specified value is not a member of the enumeration, a compile-time error is generated. Otherwise, the actual value of the enumeration is made into a constant, and a C# enumeration literal is created. A new C# enumeration literal must be created as opposed to just creating a new literal with the value’s type. If the latter method were used, it would be possible to assign one enumeration value to another enumeration value, even if the enumerations were not of the same type. By leaving the enumeration value as a true enumeration constant, the compiler will prevent cross enumeration assignment.

```
Tuple {name: String = 'Dave', age: Integer = 25}
```

Figure 5.17: *Tuple literal expression*

Figure 5.17 illustrates an example of a literal tuple. Each tuple literal is resolved by placing the body of the tuple into three lists: element names, element types, and element values. The first check is to make sure that the tuple does not contain two elements with the same name. If two elements have the same name then the tuple is considered invalid, and a compile-time error is generated. The next step attempts to resolve each of the element values as a constant. If the tuple contains a value that cannot be resolved as a constant, the whole tuple cannot be resolved as a constant. A non-constant tuple is not an error. The tuple will be resolved during the resolution of non-constant OCL expressions. If each of the tuple values can be resolved into a constant, then the value types are checked. Checking the value types consists of checking the type of the element’s value against the type specified in the tuple literal. The specification of an element type in a tuple literal is optional. In this case, the type from the element’s value is used. Once the tuple element types are verified, the information gathered in the three lists is used to generate a C# expression that will construct a new instance of the “Tuple” class defined in the OCL type library. Figure 5.18 shows the C# code generated from the tuple literal shown in Figure 5.17.

Collection literals are resolved similarly to the tuple literal. Figure 5.19 provides some example collec-

```

new OCL2STL.Tuple(new Object[] { "name", "age" },
    new Object[] { "Dave", 25 },
    new Object[] { OCL2STL.OclString, OCL2STL.Integer });

```

Figure 5.18: C# code to represent a tuple literal expression

tion literals. The first step in resolving a collection literal is to attempt to resolve each of the elements of the collection into a constant. If any of the elements cannot be resolved into a constant, then the entire collection cannot be resolved into a constant. A compile-time error is created if the collection cannot be resolved into a constant. The collection will be resolved during the resolution of non-constant expressions. In the case where the “..” operator is found, the compiler will insert the missing elements into the collection and resolve them. Once each of the elements contained within a collection is resolved into a constant, the types of the elements are compared to ensure that each element of the collection is of the same type. If an element is of a different type than the other elements in the collection, a compile-time error is generated. In the case where the collection type is a “Set” or an “OrderedSet”, the elements are checked to ensure that no duplicates exist. If a duplicate does exist, a compile-time error is generated. With the resolution of the collection elements complete, the collection and its elements are mapped to a C# expression. The C# expression will create a new instance of the corresponding C# class that was defined in the OCL type library to represent the OCL collection type. Figure 5.20 shows the resulting C# code for the collection literals presented in Figure 5.19.

```

Set {1, 2}
Sequence {1..(2+3)}
Bag {Set {1, 2}, Set {3, 4}}
OrderedSet {}

```

Figure 5.19: Collection literal expressions

```

new OCL2STL.Set(new Object[] { 1, 2 });
new OCL2STL.Sequence(new Object[] {1, 2, 3, 4, 5});
new OCL2STL.Bag(new Object[] {
    new OCL2STL.Set(new Object[] {1, 2}),
    new OCL2STL.Set(new Object[] {3, 4})});
OCL2STL.OrderedSet.EmptyOrderedSet();

```

Figure 5.20: C# code to represent collection literal expressions

The final type of literal expression is a void literal. The void literal is denoted by the use of the “OclVoid” type outside of a context expression. The “OclVoid” is used in an OCL expression. As previously stated, the “OclVoid” type is mapped onto the *null* keyword found in C#. When a void literal is encountered in an OCL expression, it is mapped to a null C# literal.

Attribute Call Expressions

There are several types of attribute call expressions. Only one of these types can be resolved into a constant. An attribute call expression can only be resolved into a constant in the case where we are referencing an attribute defined on a constant tuple. Figure 5.21 shows such an expression. When such an expression is encountered, the resolution starts by resolving the left side of the expression into constant. The left side of the expression is a tuple literal. The resolution of a tuple literal will produce a C# expression that creates a new tuple object. In order to access the attribute specified by the right side of the expression, we need to execute the C# expression and create an actual tuple object at compile-time. The object is needed at compile-time so that the attribute call can be made and the result converted into a C# expression. Once the instance of the “Tuple” class is created, the attribute name is used as a key in the hash table that contains the values. If the name does not exist, then the attribute being used does not exist in the tuple, and a compile-time error is generated. Once the value of the attribute is accessed, a C# expression is generated using the value only. This means, the actual tuple and the attribute call are removed and only the result of the attribute call is converted into a C# expression. In the case of Figure 5.21, the entire OCL expression is mapped into a C# integer constant with a value of 25.

```
Tuple {name: String = 'Dave', age: Integer = 25}.age
```

Figure 5.21: Constant attribute call expression

Variable Expressions

Variable expressions are used to represent any type of name for the reasons discussed in Chapter 3. As previously discussed, these variable names may not necessarily be references to actual variables. They may be referencing other named elements.

The first type of variable expression that can be resolved into a constant is an enumeration. As an enumeration consists of a name, the parser will assign an enumeration found in an OCL expression to a variable expression instead of an enumeration literal. If the name used in the variable expression is in reference to an enumeration, the compiler will replace the variable expression with an enumeration

literal expression, and use the rules for resolving an enumeration literal for processing the variable expression.

The second type of variable expression that can be resolved into a constant is a reference to a constant value. In this case, the variable name references a previously defined constant. The compiler will attempt to locate the constant specified by the variable expression. If a suitable constant can be located, the compiler will map the OCL variable expression to a C# literal that represents the actual value of the constant.

The third and final type of variable expression that can be resolved into a constant is the use of the “OclVoid” type. The resolution of the “OclVoid” type has already been discussed under literal expressions. However, as the “OclVoid” type is represented by the name “OclVoid”, the parser will treat it as a variable expression. In the case where the name of the variable in the expression is “OclVoid”, the compiler will replace the variable expression with a null literal expression.

If the variable expression does not fall into one of the three aforementioned categories, the compiler will try to resolve the variable during the non-constant resolution step.

Operation Expressions

The final category of expressions that can be resolved into a constant are operation expressions. Operation expressions represent an operation invoked on an object. In the case of C#, an operation expression will result in the calling of a method defined on either an instance of a class or the class itself. Operations that are defined on types defined by the OCL type library can be resolved into a constant if the object that the method is being invoked on is a constant. If the object can be constructed at compile-time, and the operation being invoked can be invoked at compile-time, the result of the operation will be used as the C# expression. If either the object or the operation cannot be used at compile-time, the operation expression cannot be resolved into a constant, and will be resolved during non-constant resolution.

There are two types of operation expressions: operations using operators, and operations that do not use operators. Figure 5.22 provides a sample of both types of expressions. Regardless of the operation

expression type, the first step is to resolve the left side of the expression into a constant. In the case of an expression using an operator, the left side of the expression is the first operand. In the case of the first expression in Figure 5.22, the left side of the expression is “3”. In the case where there is only one operand, that operand will be used as the left side. If the left side of the expression cannot be resolved into a constant, the operation expression will be resolved during non-constant expression resolution.

```

3 + 5
(3 + 5) * 2
3 + 5 * 2
Sequence {1..10}.size()
-24.abs()
'Dave'.size()
not true

```

Figure 5.22: Constant operation expressions

Operators

With the left operand of the expression resolved into a constant, the next step is to look at the operator that is being applied. One of the more interesting operators is the parenthesis “()”. The parenthesis can be used as an operator to indicate precedence. The handling of precedence is done by the parser. When we encounter parenthesis during the semantic pass, the operator has already been applied. The precedence is reflected in the structure of the AST. Thus, when the parenthesis is encountered, the contents of the parenthesis are resolved and used as the expression. In terms of constant resolution, a call to the parenthesis operator is seen as a constant, if the contents can be resolved as a constant. If this is not the case, the operation expression will be resolved as a non-constant.

If the operator is determined not to be the parenthesis, the next step is to create an instance of the type that the left side of the expression resolved to. In the case where the left side of the expression is a literal, an instance of the literal’s type is created. In the case of the first expression of Figure 5.22, an instance of the “Integer” class is created and assigned a value of 3.

The OCL defines two types of operators: unary and binary. Unary operators only contain one operand. Binary operators use two operands. The OCL specification defines two unary operators: “-” and “not”. The operator “-” can be applied to the “Integer” and “Real” types only. The “not” operator only applies

to “Boolean” types. When a unary operator is encountered on a valid type, the newly created compile-time instance is used, and the operator is applied to the instance at compile-time. The result of applying the operator at compile-time is then used to construct a C# expression. For example, consider the OCL expression: “not true”. The first step is the “true” value is determined to be the left side of the expression and is resolved into a literal of the “Boolean” type. An instance of the “Boolean” type that was defined in the OCL type library is created, and assigned the value of the left side of the expression: “true”. The next step is to apply the “not” operator to the instance that was created. Finally, the new value of the instance, “false”, is used to generate a C# boolean literal expression.

Binary operators are resolved in a similar fashion as the unary ones. Unlike unary operators, binary operators require two operands. The left side of the operand has already been resolved. However, the right side still needs resolution. If the right side of the expression cannot be resolved into a constant, the resolution of the operation call expression will be done as a non-constant. In order for a binary operator to be resolved as a constant, both of the operands must be able to be resolved as a constant. Before the operator is applied, the types of the two operands are checked to see if there is a corresponding operator defined that supports the two given types. If the types do not match the operator, a compile-time error is generated. With both of the operands resolved and checked, each is assigned to a new instantiation of their type. The binary operator is applied using the two compile-time instances to produce a result. The result is then converted into a C# expression.

Non-Operators

In the case where an operator is not used, an operation call expression consists of an operation being called on an object or type. In the case of constant resolution, only operations defined on the types defined in the OCL type library can be used. As there is no class level operations defined, only operations on objects are expected.

The already resolved left side of the expression represents the object. The next step is to determine if the operation being invoked in the OCL expression is defined by a method in the C# implementation of the OCL type. In the case where a corresponding C# method does not exist, a compile-time error is

generated. The error indicates that the requested operation is not defined. Depending on the operation, there may be arguments. If the correct number of arguments are present, each of the arguments are resolved into a constant. If any of the arguments cannot be resolved into a constant, the entire operation call expression cannot be resolved into a constant. With all of the arguments resolved into constant values, the left side of the expression as well as each of the arguments is instantiated into objects of their respective types. The requested operation is invoked via the corresponding method defined on the left side's instance using the given arguments. The result of the method call is then converted into a C# expression. Figure 5.23 shows the result of resolving the OCL expressions found in Figure 5.22.

```
8
16
13
10
24
4
false
```

Figure 5.23: *Resolved constant operation expressions*

Constant Resolution Summary

An expression can be resolved into a constant value if all of the components of the expression can be resolved into a constant. The purpose of the constant resolution process is to reduce the amount of processing done at runtime. By converting complex constant expressions into a single value, the runtime is not required to compute the same value repeatedly. As well, the developer is free to use any type of calculation expression without having to worry about performance. For example, the expression “ $5 + 6 * 3$ ” requires no more runtime computation than the expression “23”.

It is straightforward to see that not all expressions can be resolved into a constant value. Some expressions require the use of runtime values to determine their value. The next section will look at the resolution process used to resolve an OCL expression if it cannot be resolved into a constant value using the previously stated mappings.

5.3.2 OCL Expression Resolution

Before an OCL expression can be used in C#, the OCL expression must be fully resolved, and mapped into a C# expression. Some of the C# expressions consist of simple value types, others consist of a series of C# expressions. The first step in resolving each OCL expression is to try to resolve the expression into a constant value. If the expression cannot be resolved into a constant value, then the main OCL resolution process is executed. The following sections will examine how each of the expression types discussed in Chapter 3 are resolved into C#. The order in which the sections are presented is the order in which they are resolved in the implementation. The order of resolution is important in order to resolve some of the ambiguities discussed in Chapter 3.

Let Expressions

A let expression enables the definition of a variable. The variable is assigned via an OCL expression. When the variable is used in future OCL expressions, the variable is replaced by the OCL expression that is assigned to it. Figure 5.24 illustrates an example let expression. Figure 5.24 defines a let expression with a variable named: “workingSet”. The “workingSet” variable is assigned to be a set of integers.

```
class MyClass
{
    OCL
    [
        "Context MyClass::Method1
        (a : Integer, b : Integer, c : Integer) : OclVoid"
        "pre: let workingSet : Set(Integer) = Set { a, b, c }"
        "in"
        "workingSet.size() = 3 and workingSet.count(a) = 1"
    ]
    void Method1(int a, int b, int c)
    {
    }
}
```

Figure 5.24: *A let expression*

Analogous to the definition of a let expression, the variables defined in a let expression are each represented by a local variable in C#. The resolution of a let expression begins with the resolution of the variable’s expression. With the variable’s expression resolved, the type of the expression is compared with the given variable type. If the types do not match, a compile-time error is generated. After the types

are verified, a new local variable is added to the body of the C# element where the let expression is being used. In order to avoid conflict with existing local variables, the C# local variable name is defined as the OCL let variable prefixed with “ocl_let_”. In the case of Figure 5.24, the C# variable name would be “ocl_let_workingSet”. A map between the OCL variable name and the C# variable name is then created and saved. Finally, the remainder of the OCL expression is resolved. If the let variable name is encountered, the variable is replaced with a reference to the C# variable name. Figure 5.25 shows the C# code generated by the OCL let expression from Figure 5.24. The value of the “ocl_pre_1” variable will be used later to determine if the precondition has been violated.

```
bool ocl_pre_1 = false;
Set ocl_let_workingSet = new Set(new Object[] {a, b, c});
ocl_pre_1 = ocl_preocl_let_workingSet.size() == 3 &&
    ocl_let_workingSet.count(a) == 1;
```

Figure 5.25: C# code to represent a let expression

If Expressions

An if expression allows different OCL expressions to be executed depending on a condition. The format of an OCL if statement is illustrated by Figure 5.26. Each of the three components of the if expression are complete OCL expressions. An if expression is resolved by resolving each of the three components independently. After the three components are resolved into C# expressions, the expressions are used to construct a C# conditional statement, as shown in Figure 5.27.

```
if <Condition>
then <Condition true expression>
else <Condition false expression>
endif
```

Figure 5.26: An if expression

```
(<Resolved condition>) ? <Resolved true expression> :
    <Resolved false expression>;
```

Figure 5.27: C# code to represent an if expression

Iterator Expressions

The OCL defines a number of standard OCL operations that enable the elements of a collection to be iterated through and for each element: evaluate an expression on that element. As shown in Figure 5.28, each iterator may have an extra optional parameter. The optional parameter is called an iterator variable. An iterator variable is a variable that can be used within the iterator's body expression to reference the element of the collection that is currently being evaluated. The iterator expression in Figure 5.28 states that each of the employees in a company must have a unique social insurance number.

```
context Company
inv: self.employees->isUnique(p : Person | p.sinNumber)
```

Figure 5.28: *An iterator expression*

Table 5.2 lists each of the iterator operations that are defined on all of the collection types.

Operation	Description
any	Returns a random element of the source collection for which the body expression is true.
collect	Returns the collection of objects that result from evaluating the body expression for each element in the source collection.
collectNested	Returns a collection of collections that result from evaluating the body expression for each element in the source collection.
exists	Returns true if there is at least one element in the source collection for which the body expression is true.
forAll	Returns true if the body expression is true for all elements in the source collection.
isUnique	Returns true if the body expression yields a different value for each of the elements in the source collection.
one	Returns true if there is exactly one element in the source collection for which the body expression is true.
reject	Returns a collection of elements for which the body expression is false.
select	Returns a collection of elements for which the body expression is true.
sortedBy	Returns a collection containing all of the elements of the source collection ordered by the body expression.

Table 5.2: *Iterator operations defined on all collection types*

As iterators are complex in nature, the resolution process consists of several steps.

The first step in resolving an iterator expression is to resolve the left side of the expression. That is, resolve the side of the expression that the iterator is being applied. There are two cases following

successful resolution of the left side of the expression. The first is that the left side of the expression resolves into a collection type. In this case the resolution process continues, as there is a valid collection type to which the iterator can be applied. The second case is when the left side of the expression does not resolve into a collection type. In this case, the user is trying to apply an iterator expression to a non-collection type. The OCL specification states, that if an iterator expression is applied to a non-collection type, a new collection of type “Set” is created. The set will contain the left side of the expression that could not be resolved into a collection type.

Once the left side of the iterator expression is either resolved into a collection type, or encapsulated within a collection type, each of the variables defined within the iterator are examined. For each of the variables defined within the iterator, the variables type is compared against the source collection’s element type. Each of the variable types must match the collection element type. If a match cannot be made, a compile-time error is generated.

After the iterator variable types are matched to the collection element type, the variable names are stored in a cache. The cache is needed so that if the variable names are referenced in the iterator’s body, they can be resolved into a C# variable reference expression.

With the variable names stored for the resolution of the iterator’s body, the body expression is resolved using the resolution rules defined in this chapter. With each of the iterator variables processed, and the body expression resolved, the actual iterator can now be resolved. Each of the predefined iterators defines a different operation and thus has a slightly different resolution procedure. The next sections will examine the resolution of each of the predefined iterators in detail.

Any

The “any” iterator returns a random source collection element for which the body expression yields a value of true. The first step in resolving the “any” iterator is ensuring that the body expression will yield a result that is of the “Boolean” type. If the body expression does not have such a result, a compile-time error is generated. The next step is to ensure that the “any” iterator does not contain more than one

iterator variable. The “any” iterator is not defined to operate on more than one iterator variable[32]. The use of more than one iterator variable results in a compile-time error.

Once the body expression and iterator variables are verified to be compatible with the “any” iterator, the C# code generation process begins. The first step in the process is to create a temporary variable to store the result of the iterator. The result variable is assigned an initial value of null. Once the result variable has been initialized, a C# *foreach* statement is used to iterate through each element in the collection. The body of the *foreach* statement consists of a single statement for each of the previously defined iterator variables. Each of the variables is assigned the value of the element that is currently being iterated through. Following the initialization of the iterator’s variables, the C# expression that was generated as a result of the resolution of the body expression is inserted. The body expression is used in the context of an *if* statement. If the execution of the body statement results in true, then the element that is currently being iterated is assigned to the result variable.

Upon completion of the *foreach* statement, the result variable is either assigned to one of the elements in the collection that satisfies the body expression, or its initial value of null. Figure 5.29 illustrates the pseudo code for the resolution of the “any” iterator.

```
<CollectionElementType> iterator_result := null
foreach(<CollectionElementType> iterator_element in source)
  <CollectionElementType> iterator_variable := iterator_element
  if(<BodyExpression> = true)
    iterator_result := iterator_element
  break
```

Figure 5.29: Pseudo code for the resolution of the any iterator

Collect

The “collect” iterator results in a collection of objects that result from evaluating the body expression on the source collection. The first step in resolving the “collect” iterator is to make sure that not more than one iterator variable exists. If more than one iterator variable exists, a compile-time error is generated. Once the number of iterator variables is checked, a collection to store the results is created. The result collection type depends on the type of the source collection. If source collection is of type “Sequence”

or “OrderedSet”, the result collection is of type “Sequence”, otherwise the result collection is of type “Bag”[32].

Once the type of the result collection is determined, the C# code generation process begins by creating a new local variable to store the result. The local variable is assigned an initial value of an empty collection. The collection type is the result collection type determined before the code generation process began.

Following the initialization of the result variable, a *foreach* statement is used to iterate through each element of the collection. As with the “any” iterator, the body of the *foreach* statement begins with the assignment of the current collection element to the previously defined iterator variables. Once the iterator variables have been assigned, a statement that inserts a new collection element into the result collection is added. The value of the element that is inserted is defined by applying the iterator’s body expression to the current collection element being iterated.

Once the body of the *foreach* statement is defined, a final statement is required to complete the resolution of the iterator. The statement applies the “flatten” collection operation to the resultant collection. The “flatten” operation removes any nested collections, so that the resultant collection only contains actual collection elements, and not nested collections of elements. For more information on the “flatten” operation, please see [32]. Figure 5.30 illustrates the pseudo code for the resolution of the “collect” iterator.

```
<ResultCollectionType> iterator_result :=
  <ResultCollectionType>.EmptyCollection()
foreach(<CollectionElementType> iterator_element in source)
  <CollectionElementType> iterator_variable := iterator_element
  iterator_result := iterator_result.including(<BodyExpression>)
iterator_result := iterator_result.flatten()
```

Figure 5.30: Pseudo code for the resolution of the collect iterator

Collect Nested

The “collectNested” iterator performs the same operation as the “collect” iterator with one exception: the “collect” iterator flattens the result collection, where the “collectNested” iterator does not. The only

difference between the pseudo code shown in Figure 5.30, and the pseudo code for the “collectNested” iterator is that the pseudo code for “collectNested” does not contain the last line. As this is the only difference between the two iterators, the “collectNested” iterator will not need to be discussed further.

Exists

The “exists” iterator returns true if there is at least one element in the source collection that the body expression results in a true value. The first step in resolving the “exists” iterator is to verify that the body expression will result in a boolean value. If the body expression does not yield a result of type “Boolean”, the body expression does not conform to the specification of the “exists” iterator, and a compile-time error is generated. Once the body expression is verified, the iterator variables are also checked to ensure that there is no more than one iterator variable. If more than one iterator variable is present in an “exists” iterator, a compile-time error is generated.

After the body expression and number of iterator variables are verified, the C# code generation process begins. The first step is the introduction of a local variable to store the result of the iterator. The variable will be of type “Boolean”. Once the variable has been added, it is assigned an initial value of false.

Following the setup and initialization of the result variable, a *foreach* statement is added to iterate through each element in the source collection. The body of the *foreach* statement consists of the assignment of the current element that is being iterated, to the variables defined in the iterator. After the assignment of the iterator variables, an assignment statement is added to complete the body of the *foreach* statement and complete the iterator. This assignment statement assigns the result variable. The result variable will hold its current value bitwise “or”ed with the value of the body expression. Figure 5.31 presents the pseudo code for the “exists” iterator. Upon completion of the *foreach* statement, the result variable contains the result of the iterator.

```
OCL2STL.Boolean iterator_result := false
foreach(<CollectionElementType> iterator_element in source)
    <CollectionElementType> iterator_variable := iterator_element
    iterator_result := iterator_result | <BodyExpression>
```

Figure 5.31: Pseudo code for the resolution of the exists iterator

For All

The “forAll” iterator returns true if the body expression evaluates to true for each element in the source collection. The “forAll” iterator operates similarly to the “exists” iterator. The obvious difference is that the “forAll” iterator uses a bitwise “and” instead of a bitwise “or”. There is one additional difference between the “forAll” iterator and the “exists” iterator: the “forAll” iterator may define up to two iterator variables.

In the case where only one iterator variable is defined, the “forAll” iterator behaves the same as the “exists” iterator. In the case where two iterator variables exist, more differences exist.

Where two iterator variables are defined, the code generation process begins with the introduction of the same result variable, with the same initial value: false. Following the result variable setup, a *foreach* statement is inserted. The body of the *foreach* statement consists of the initialization of the first iterator variable, followed by a second *foreach* statement. The second *foreach* statement’s body begins with the initialization of the second iterator variable. Once the second iterator variable has been initialized, the final assignment statement is added. The final assignment statement is the same as the one used in the “exists” iterator, except a bitwise “and” is used. Figure 5.32 illustrates the pseudo code for a two variable “forAll” iterator.

```
OCL2STL.Boolean iterator_result := false
foreach(<CollectionElementType> iterator_element_1 in source)
  <CollectionElementType> iterator_variable_1 := iterator_element_1
  foreach(<CollectionElementType> iterator_element_2 in source)
    <CollectionElementType> iterator_variable_2 :=
      iterator_element_2
    iterator_result := iterator_result & <BodyExpression>
```

Figure 5.32: Pseudo code for the resolution of the *forAll* iterator

Is Unique

The “isUnique” iterator returns true if the body expression yields a different value for each element in the source collection. The resolution of the “isUnique” iterator begins with checking the number of iterator variables. If there is more than one iterator variable, a compile-time error is generated. Once the number of iterator variables has been verified, the C# code generation process begins.

With the other iterators, a result variable is inserted to store the result of the iterator. The result variable in the case of the “isUnique” iterator has a type of “Boolean”, and an initial value of false. In addition to the result variable, execution of the “isUnique” iterator requires a temporary collection to store the result of executing the body expression on each of the elements of the source collection. The collection will be represented by a C# “ArrayList” rather than an OCL collection. The main driver for not using an OCL collection type is that the special functionality provided by the OCL collection types is not needed.

Once the result and temporary collection variables have been initialized, a *foreach* statement is inserted to iterate through each element in the source collection. The body of the *foreach* statement consists of the assignment of the collection element that is currently being iterated to the iterator variable. Once the iterator variable has been assigned, the iterator’s body statement is used to determine if the value of the body statement applied to the element being iterated is already in the temporary collection. If the value is not already in the temporary collection, it is added.

Upon the completion of the *foreach* statement, the result variable is assigned true if the temporary collection contains the same number of elements as the source collection; false otherwise. If the body of the iterator produces a unique value for each of the elements of the collection, the number of unique values will equal the number of elements in the source collection. The pseudo code for the execution of the “isUnique” iterator is shown in Figure 5.33.

```
OCL2STL.Boolean iterator_result := false
System.ArrayList temp_collection := new System.ArrayList()
foreach(<CollectionElementType> iterator_element in source)
    <CollectionElementType> iterator_variable := iterator_element
    if(not temp_collection.Contains(<BodyExpression>))
        temp_collection.Add(<BodyExpression>)
iterator_result := (source.size() = temp_collection.Count)
```

Figure 5.33: Pseudo code for the resolution of the *isUnique* iterator

One

The “one” iterator returns true if there is exactly one element of the source collection that results in a true evaluation of the body expression. The resolution and execution of the “one” iterator is similar to that of the “isUnique” iterator. The “one” iterator only accepts at most one iterator variable and uses a

temporary collection to store the source collection elements that, when applied to the body expression, yield a true result. Following the execution of the *foreach* statement, the result of the iterator is calculated by checking that the temporary collection only contains one element. This implies, the body expression only yields a true value for one element in the source collection. Figure 5.34 shows the corresponding pseudo code.

```

OCL2STL.Boolean iterator_result := false
System.ArrayList temp_collection := new System.ArrayList()
foreach(<CollectionElementType> iterator_element in source)
  <CollectionElementType> iterator_variable := iterator_element
  if(<BodyExpression>)
    temp_collection.Add(iterator_element)
iterator_result := (temp_collection.Count = 1)

```

Figure 5.34: Pseudo code for the resolution of the one iterator

Reject

The “reject” iterator returns a collection that contains all of the elements from the source collection that, when evaluated with the body expression, resulted in false. The resolution of the “reject” iterator begins with determining the type of the result collection. The result collection will store elements of the same type as the source collection. If the source collection is a “Set” or “OrderedSet” type, the result collection will be of type “Set”. The “Bag” and “Sequence” collection types result in a collection of type “Bag” and “Sequence”, respectively. The iterator variables are checked to see that no more than one is specified. More than one iterator variable results in a compile-time error. Finally, the body expression is checked to ensure that it returns a result of type “Boolean”. If this is not the case, a compile-time error is generated.

With the type of the result collection determined, a result variable is defined using the result collection type. The result variable is initialized to an empty collection of the corresponding type. Following the initialization of the result collection, a C# *foreach* statement is added. The body of the *foreach* statement consists of the assignment of the iterator variables. The iterator variables are assigned to the element from the source collection that is being iterated.

Following the assignment of the iterator variables, an *if* statement is used to determine if the execution of the body statement results in a true or false value. If the body statement produces a false value,

the source collection element that produced the false value is added to the result collection. Once the *foreach* statement has executed, the result of the iterator is the collection contained in the result variable.

Figure 5.35 illustrates the pseudo code for the “reject” iterator.

```

<ResultCollectionType> iterator_result :=
  <ResultCollectionType>.NewCollection()
foreach(<CollectionElementType> iterator_element in source)
  <CollectionElementType> iterator_variable := iterator_element
  if(not <BodyExpression>)
    iterator_result := iterator_result.including(iterator_element)

```

Figure 5.35: Pseudo code for the resolution of the reject iterator

Select

The “select” iterator is the inverse of the reject iterator. The “select” iterator results in a collection of elements from the source collection, where the body expression results in true. The only difference in terms of resolution or C# code generation is that the *if* statement shown in Figure 5.35, does not contain the “not” operator. Otherwise, the “select” iterator follows the same rules as outlined during discussion of the “reject” iterator.

Sorted By

The “sortedBy” iterator returns a collection containing all of the elements in the source collection ordered by the result of the body expression. The resolution and C# implementation of the “sortedBy” iterator is by far the most complex. The resolution begins with checking to make sure that no more than one iterator variable has been defined. The “sortedBy” iterator is defined to accept, at most, one iterator variable. If more than one iterator variable is provided, a compile-time error is generated. Once the number of iterator variables is verified, the resulting collection type is determined. The result collection will contain elements of the same type as the source collection. If the source collection is of type “Set” or “OrderedSet”, the resultant collection will be of type “OrderedSet”. If the source collection is of type “Bag” or “Sequence”, the resultant collection will be of type “Sequence”. The resultant collection type is a collection type that represents element order as a property of the collection, because the “sortedBy” iterator results in an ordered collection of elements.

Once the number of iterator variables and the result collection type has been determined, the C# code generation process can begin. The first step is to create and initialize the result collection variable. The result collection has an initial value of a new and empty collection of the corresponding collection type.

In addition to the result collection, two temporary collections are required during the execution of the “sortedBy” iterator. These two temporary collections are of type “ArrayList”, and begin with an initial value of an empty list. After the result collection and temporary collections have been initialized, a *foreach* statement is added to iterate over the elements of the source collection. The body of the *foreach* statement begins with the assignment of the iterator variables. Each of the iterator variables are assigned the value of the element that is being iterated. Following the assignment of the iterator variables, a statement is added to evaluate the body expression on the element being iterated. The result of the iteration and the element that was used to generate the result are added to the two temporary collections, respectively.

Following the *foreach* statement, a new temporary collection is used to store the elements of the source collection in sorted order. The new collection is given an initial value of an empty collection, as elements will be added to the collection as needed.

After the new collection is created, a *for* loop is used to loop once for each element in the source collection. For each element in the source collection, the temporary collections are used to determine where in the resultant collection the current element belongs. The element is inserted into the third temporary collection in its correct sorted location.

Once the *for* loop has been completed, the elements in the third collection are copied into the resultant collection. The copying is required so that the resultant collection is of the correct type, and not the C# “ArrayList” type. The pseudo code for the “sortedBy” iterator is shown in Figure 5.36.

Figure 5.36 shows the use of two “ArrayList” objects to implement a hash table. An actual hash table object cannot be used because there is a possibility of duplicate keys.

The explanation of the “sortedBy” iterator completes the resolution of the predefined iterators. The OCL also defines a general-purpose iterator. General-purpose iterator expressions are processed by the parser to create the next category of expressions: iterate expressions.

```

<ResultCollectionType> iterator_result :=
  <ResultCollectionType>.NewCollection()
System.ArrayList temp_array_1 := new System.ArrayList()
System.ArrayList temp_array_2 := new System.ArrayList()
foreach(<CollectionElementType> iterator_element in source)
  <CollectionElementType> iterator_variable := iterator_element
  temp_array_1.Add(<BodyExpression>)
  temp_array_2.Add(iterator_element)
System.ArrayList temp_array_3 := new System.ArrayList()
foreach(<CollectionElementType> iterator_element in temp_array_2)
  Integer pos := temp_array_3.Count
  for(integer i := 0; i < pos; i := i + 1)
    <BodyExpressionType> value :=
      temp_array_1[i] as <BodyExpressionType>
    if(value < <BodyExpression>)
      i := pos
      break
  temp_array_3.Insert(pos, iterator_element)
iterator_result :=
  new <ResultCollectionType>(temp_array_3.ToArray())

```

Figure 5.36: Pseudo code for the resolution of the *sortedBy* iterator

Iterate Expressions

Iterate expressions are generated by the parser to represent a single OCL expression: the iterate operation. The iterate operation is the most fundamental and complex of all of the iterator operations[45]. The iterate operation is also the most generic iterator. All of the iterators presented in the previous sections can be expressed in terms of the iterate operation. Figure 5.37 illustrates the syntax of the iterate operation.

```

collection->iterate(element: Type1; result : Type2 =
  <InitialExpression> | <BodyExpression>)

```

Figure 5.37: An iterate expression

The element variable is the iterator variable. The result variable is used to accumulate the result of the iterator. The initial value of the result is given by the initial expression. None of the parameters in Figure 5.37 are optional. The result of the iterate operation is the value, which is obtained by iterating over each of the elements in the source collection. For each element in the source collection, the body expression is calculated using the previous value of the result variable. Figure 5.38 shows a simple iterate expression, which results in the sum of a set of integers.

```

Set {1, 2, 3}->iterate( i : Integer, sum : Integer = 0 | sum + i)

```

Figure 5.38: An iterate expression to calculate the sum of a set of integers

In terms of resolution, iterate expressions are resolved in a similar manner to iterator expressions. However, there are some differences. The resolution process begins with ensuring that the source collection is actually a collection. In the case where the source type is not a collection type, a new collection of type “Set” is used to encapsulate the non-collection type.

Once the object that the iterate operation is being applied to is transformed into a collection type, if needed, the iterator variable is examined. The type of the iterator variable must match the collection element type. If this is not the case, the iterator cannot function because the collection elements cannot be assigned to the iterator variable. If the types do not match, a compile-time error is generated. Following the verification of the iterator variable, the result variable is checked to ensure that the variable type matches the resulting type of the initial expression.

Once the two variables are verified to be correct, the variable names and types are stored in a cache. As with the iterator expressions, the variables need to be stored so that during the resolution of the body expression, the variables can be resolved correctly.

With the variables stored, the body expression is resolved. Upon successful resolution of the body expression, the type of the body expression is compared with the type of the result variable to ensure compatibility. If the variable types do not match, a compile-time error is generated. Once the body expression type has been checked against the result variable, the C# code generation process begins.

The code generation begins with the addition of the result variable, and the assignment of the initial expression to the result variable. Following the initialization of the result variable, a *foreach* statement is used to represent the actual iteration. The body of the *foreach* statement begins by assigning the element of the source collection that is currently being iterated to the iterator variable. The body of the *foreach* statement is completed by adding an assignment statement, which assigns the value of the body expression to the result variable.

Following the execution of the *foreach* statement, the result of the iterate operation is the result variable. The resultant type of an iterate operation is the type of the result variable. Figure 5.39 shows the pseudo code for an iterate operation.


```

<ResultType> iterator_result := <InitialExpression>
foreach(<CollectionElementType> iterator_element in source)
  <IteratorVariableType> iterator_variable := iterator_element
  iterator_result := <BodyExpression>

```

Figure 5.39: *Pseudo code for the resolution of an iterate expression*

Attribute Call Expressions

The remainder of the OCL expression resolution process involves the resolution of attribute call expressions, operation expressions, variable expressions, and literal expressions. Each of these four expression types have already been resolved in the context of the constant expression resolution discussed in Section 5.3.1. The additional resolution presented here is in addition to the constant resolution. The resolution that takes place here handles the non-constant resolution operations. This resolution takes place following a failed constant resolution.

In terms of attribute call expressions, the constant resolution process only handles the case where an attribute call is being performed on a constant tuple. The following sections will illustrate the resolution process for the remaining types of attribute call expressions. The first step in resolving any type of attribute call expression is to resolve the left side of the expression. The left side of the expression is the class or object to which the attribute is being called on. After resolving the left side of the expression into a valid object, the following types of attribute call expressions are resolved.

Static Attributes

If the left side of the expression resolves into a class, rather than an object, the attribute call is being made in the context of a static member. Figure 5.40 illustrates such a case. The first step in resolving the attribute call is to determine the type information of the left side of the expression. The type information consists of all members and operations defined on the type.

Once the type information is determined, the compiler will check to see if any static constants defined on the left type match the given attribute name. If a corresponding constant cannot be found, the search continues with the static fields, and static properties.

```
class Person
{
    public static int nextPersonNumber = 1;
    OCL
    {
        "context Person::GetNextNumber() : Integer"
        "post: result = Person::nextPersonNumber"
    }
    public int GetNextNumber()
    {
        return Person.nextPersonNumber;
    }
}
```

Figure 5.40: *A static attribute expression*

If a corresponding static member is located during the search, the compiler converts the attribute call expression into a C# member access on the type derived from the left side of the expression. If a corresponding member cannot be located during the search, a compile-time error is generated.

Self Expressions

Under attribute call expressions, a self expression is used to make an attribute call on the current object. The OCL expression that results in this case has the form: “self.attribute”. The self portion of the expression is resolved when the left side of the expression is resolved. The result of resolving the *self* keyword, is the use of the C# *this* keyword. In order to complete the resolution, an attribute located on the containing object must be located.

The resolution begins with retrieving the type information for the containing type. Once the type information is retrieved, the compiler searches through the constants, fields, and properties. If a member is found with the same name as the attribute, the attribute call expression is resolved into a C# member access applied to the containing object, via the *this* keyword.

Tuples

Under constant resolution, tuples that were constant were able to have attribute call expressions resolved into a constant. In the case where the tuple itself is not a constant, an attribute call cannot be resolved into a constant. The resolution of non-constant tuples takes place here.

A tuple is implemented via the C# “Tuple” class as discussed in Section 5.2.1. The compiler maintains each “Tuple” type separately. A tuple that consists of instances of a “Person” class called “person”, is not the same as a tuple that contains instances of a “Person” class called “employee”. The resolution of a non-constant tuple attribute call begins by determining if the requested attribute name exists within the “Tuple” type. Does the “Tuple” type define a “person” attribute?

If the requested attribute name does not exist within the “Tuple” type, a compile-time error is generated. If the name does exist, then C# code generation begins. As previously discussed in Section 5.2.1, the C# “Tuple” class is implemented via two hash tables. One hash table contains a map between the names and values, while the other contains a map between the names and the value types. In the context of an attribute call expression, the OCL expression has provided the name and is requesting the value. The generated C# code consists of a call into the name and value hash table to extract the value for the requested name. The information provided by the name and type hash table is also used by the compiler to determine the resultant type after the attribute call takes place.

Attribute Lookup

Attribute lookup is the most common and straightforward type of attribute call expression. An attribute lookup occurs when the left side of the expression resolves into an object. The compiler begins by retrieving the type information for the object. Once the type information has been retrieved, the compiler searches for a matching attribute in the following order: constants, fields, and properties.

If a suitable attribute can be found, the attribute call expression is resolved into a C# member access on the object. If an attribute cannot be found, a compile-time error is generated, and the resolution fails.

Shorthand Collect

The shorthand collect notation is used to represent the use of the “collect” iterator. Consider Figure 5.41, where there is a collection of objects of type “Employee”. The “Employee” type defines an integer field called “sinNumber”. The “collect” iterator in Figure 5.41 creates a new collection that contains that

value of the “sinNumber” field for each of the “Employee” objects in the collection. Following the creation of the new collection, the collection is tested to ensure that each employee has a different social insurance number.

```
employees->collect(emp : Employee | emp.sinNumber)->
  isUnique(i : Integer | i)
```

Figure 5.41: Use of the collect iterator to ensure that each employee has a different social insurance number

Figure 5.42 shows the same expression using the shorthand “collect” notation. The left side of the expression is a collection, and the attribute call represents the field of the collection element type that is to be collected into a new collection. When such a condition is encountered by the compiler, the compiler resolves the expression in two steps. The first step ensures that the left side of the expression is actually a collection type. If the left side of the expression is not a collection type, then the shorthand collect notation does not apply. Once the left side has been verified as a collection, the second resolution step is executed. The second resolution step is the transformation of the attribute call expression into an iterator expression. The iterator expression is then resolved as previously discussed.

```
employees.sinNumber->isUnique(i: Integer | i)
```

Figure 5.42: Use of the shorthand collect notation to ensure that each employee has a different social insurance number

@pre Notation

The final type of attribute call expression is the use of the @pre keyword. The @pre keyword is used to indicate that the value of an attribute or association at the start of the operation is desired, as opposed to the current value[45]. The @pre keyword can only be used in postconditions. In the case where the @pre keyword is used outside of a postcondition, a compile-time error is generated. Figure 5.43 shows the use of the @pre keyword. Figure 5.43 defines part of a stack class. The pop method performs the standard stack pop operation. The postcondition states that the current size of the stack should be the size of the stack before the pop, minus one.

In terms of resolution, an attribute that contains the @pre keyword is replaced by a temporary variable. The temporary variable is then initialized at the beginning of the method or accessor to be the value of

```
class Stack
{
    OCL
    [
        "context Stack::Pop() : OclAny"
        "post: size = size@pre - 1"
    ]
    public Object Pop()
    {
    }
}
```

Figure 5.43: Use of the *@pre* keyword

the requested attribute. The temporary variable stores the value of the attribute before the body of the method or accessor is executed.

At the current stage of the compiler, the method or accessor to which the *@pre* keyword is being applied is not ready for our temporary variable to be added yet. At the resolution phase, the compiler creates a temporary variable name, and uses it in the actual expression where the *@pre* keyword was found. The variable name and an expression representing the value of the variable are stored in a cache. The cache allows other resolutions to replace other occurrences of the *@pre* keyword on the same attribute. The cache will be used later during the code generation phase to actually insert the variable. The details of which will be discussed later in this chapter.

If an attribute call expression is not resolved under one of the preceding types, the compiler generates a compile-time error indicating that the requested attribute could not be located.

Operation Expressions

Operation expressions that cannot be resolved into a constant consist of any operation defined on a non-OCL standard library type, as well as any operation called on a non-constant object. Meaning the left side of the expression cannot be resolved into a constant. There are several types of operation expressions, each of which will be examined in the following sections.

Accessor

The accessor type of operation expression occurs when the “->” operator is used as opposed to the standard “.” operator. The “->” operator is used under the context of iterators, and in the case where a non-collection type is to be used as a collection type. In Figure 5.44, the “->” operator is being used so that a person object can be treated as a collection.

```
person->size() = 1
```

Figure 5.44: *Use of the -> operator*

The resolution of an accessor expression is fairly straightforward. If the left side of the expression is already a collection type, the “->” operator has the same functionality as the “.” operator and no further work is required. In the case where the left type of the expression is not a collection type, a new collection of type “Set” is created, and the left side of the expression is used as the sole collection element. Figure 5.45 shows how the expression in Figure 5.44 is resolved.

```
Set {person}.size() = 1
```

Figure 5.45: *Resolution of the -> operator*

An accessor expression is not an operation call by itself, but rather it is the separator between the object and the operation. If the resolution of the accessor expression results in a new collection, the left side of the expression is replaced with the new collection. Following the replacement, the actual resolution proceeds.

The “OclAny” type defines several operations that can be applied to all objects in the system. Rather than having to modify each object, the compiler implements these operations in code. Each time one of these operation calls is discovered the compiler resolves the operation into C# code, rather than generating a method call.

The oclAsType Operation

The “oclAsType” operation represents a type cast. The operation takes a single parameter, which represents the name of the type that the left side of the expression is to be treated as. The resolution process

begins with the verification that only a single argument is provided. Once the correct number of arguments has been provided, the argument is resolved. If the argument cannot be resolved into a type, a compile-time error is generated.

Following the resolution of the argument, the requested type is checked against the left side type to ensure that the types are compatible. If their types are compatible, the left type can be converted into the requested type. A compile-time error is generated if the types are not compatible.

Upon successfully verifying that the two types are compatible, C# code is generated to replace the call to the “oclAsType” operation with a type cast. For example, the OCL expression: “anApple.oclAsType(Fruit)” is resolved into the following C# cast: “((Fruit)anApple”. The example assumes that “anApple” is an instance of a class that derives from the “Fruit” class; or that “anApple” is an instance of the “Fruit” class.

The oclIsTypeOf Operation

The “oclIsTypeOf” operation represents a comparison between two types. The operation is defined to return true if the type of the left side of the expression is equal to the type specified in the operation’s sole argument[32].

The resolution of the operation begins the same as the “oclAsType” operation; the number of arguments are checked, and the argument is resolved into a type. From this point, the compiler extracts the type information from the left side of the expression and compares it with the type information provided by the resolution of the argument.

As the OCL is a strongly typed language, all of the type information is available at compile-time. The compiler can determine the type information for both types, and determine if the types are equal. The result of resolving the operation is a boolean literal. If the two types are equal, the expression resolves into a true literal; false otherwise. Figure 5.46 provides an example.

```

1.ocIsTypeOf(Integer)      - Resolves into a true literal
aPerson.ocIsTypeOf(Person) - Resolves into a true literal
aPerson.ocIsTypeOf(Fruit)  - Resolves into a false literal
anApple.ocIsTypeOf(Fruit)  - Resolves into a false literal
anApple.ocIsTypeOf(Apple)  - Resolves into a true literal

```

Figure 5.46: Resolution of the *ocIsTypeOf* operation

The *ocIsKindOf* Operation

The “*ocIsKindOf*” operation represents a check between two types to see if they are compatible. The operation is defined to return true if the type of the left side of the expression is equal to the type specified in the operation’s sole argument[32].

The resolution of the “*ocIsKindOf*” operation is similar to the previous operations. Following the verification of the argument, the compiler extracts the type information from both the argument and the left side of the expression.

As with the “*ocIsTypeOf*” operation, the compiler can determine at compile-time if the two types are compatible. If the types are compatible, the expression is resolved into a true literal; false otherwise.

Figure 5.47 illustrates the use of the “*ocIsKindOf*” operation.

```

1.ocIsKindOf(Integer)      - Resolves into a true literal
aPerson.ocIsKindOf(Person) - Resolves into a true literal
aPerson.ocIsKindOf(Fruit)  - Resolves into a false literal
anApple.ocIsKindOf(Fruit)  - Resolves into a true literal
anApple.ocIsKindOf(Apple)  - Resolves into a true literal

```

Figure 5.47: Resolution of the *ocIsKindOf* operation

The *allInstances* Operation

As previously mentioned, the “*allInstances*” operation is not supported by the implementation. The rationale for not supporting the “*allInstances*” operation is both semantic and technical in nature. The details of which can be found in Chapter 7. If the “*allInstances*” operation is discovered during the resolution of an OCL expression, a compile-time error is generated.

The oclIsUndefined Operation

The “oclIsUndefined” operation is used to determine if an object is defined or not. An operation is defined if it has been assigned a value, undefined otherwise. The operation returns true if the object to which the operation is being applied is defined and false otherwise.

As previously stated in Section 5.2.1, the notion of undefined in OCL is mapped into the C# *null* keyword. Thus, an object is considered to be undefined its value is equal to null. The resolution of the “oclIsUndefined” operation is the left side of the expression being compared to a null literal. The result of the comparison will be true if the left side of the expression is null and otherwise false. For example, the expression: “aPerson.oclIsUndefined()” will result in the following C# expression: “(aPerson == null)”.

The oclIsNew operation

The “oclIsNew” operation is used only in postconditions to determine if a given object has been created during the execution of a method or accessor. Meaning that the object was undefined before the method or accessor executed and defined after.

The resolution of the “oclIsNew” operation consists of several steps. The first step checks to see if the operation is being used in the context of a postcondition. If the “oclIsNew” operation is not used in the context of a postcondition, a compile-time error is generated. Once the context of the “oclIsNew” operation has been verified, a new OCL expression is generated of the form: “<leftSide>@pre.oclIsUndefined()”. The new OCL expression is used to verify that the object was not defined before the execution of the operation.

Following the resolution of the new OCL expression, the C# code generation phase generates code to ensure that the object is defined following the execution of the operation. The C# code takes the following format: “(<leftSide> != null)”.

The final step in the C# code generation process is to create a single expression that yields the value of the “oclIsNew” operation call. Figure 5.48 illustrates the complete pseudo code.

```

<LeftSideType> pre_temp := <leftSide>
<MethodOrAccessorBody>
OCL2STL.Boolean result := (pre_temp = null) and (<leftSide> != null)

```

Figure 5.48: *Pseudo code for the resolution of the oclIsNew operation*

It can be shown that the result expression shown in Figure 5.48 does not ensure that the left side of the expression was actually created during the execution of the method or accessor body. The result expression only checks to see that the left side of the expression has been assigned a value. The compiler would have to examine the right side of the assignments to the left side of the expression during the execution of the method or accessor body. Doing so adds significant complexity and is postponed for future work.

The oclIsInState Operation

The “oclIsInState” operation results in true if the object specified by the left side of the expression is in the state specified by the given argument[32]. As discussed in Section 5.2.3, C# does not contain explicitly named states. Two objects are considered to be in the same state if they are instantiated from the same class, and the values of each of their fields are equal.

The resolution of the “oclIsInState” operation beings by checking that only one argument is provided. The argument is then resolved. The argument type is then compared with the type of the left side. If the types do not match, a compile-time error is generated, as the object cannot be used to compare state.

Once the types are verified, the C# code generation consists of an equality statement. The two objects are compared, and if found equal, are determined to be in the same state. If user defined objects are used with the “oclIsInState” operation, the user should provide an appropriate equality operator. Figure 5.49 provides an example of the “oclIsInState” operation.

```

1.oclIsInState(1) - True
1.oclIsInState(2) - False
aPerson.oclIsInState(aPerson) - True
aPerson.oclIsInState(anotherPerson) - Depends on the
    behavior of the Person::= operator

```

Figure 5.49: *Use of the oclIsInState operation*

True Operation Calls

After the special cases are handled, the remaining operation call expressions consist of two types: operators and operations. The operators are handled the same way as they are handled under constant resolution, except that unlike constant resolution, the operands may not be constants.

In terms of resolution, an operator call begins with the resolution of the left operand, and if required, the right operand. Once the two operands are resolved, the types of the operands are examined to see if the requested operator exists. If such an operator is not found, a compile-time error is generated. If a suitable operator is found, the operator call is converted into a C# invocation. The left operand is used to invoke the requested operator. If the operator is a binary operator, the right operand is used as an argument to the operator call. The result of the invocation is used as the result of the operation call.

The only remaining type of operation expression is the generic operation call. The generic operation call is when an operation is called on an object, with zero or more arguments. The first step in resolving an operation call is to resolve the left side of the expression. In the case where the left side of the expression does not exist, the containing object will be used. In terms of C#, the left side of the expression is replaced by the *this* keyword.

Once the left side of the expression is resolved or derived, the next step is to determine if the left side represents an instance or a type. If the left side of the expression represents a type, only static methods will be considered. Following the processing of the left side, the arguments, if any, are examined. Each argument is resolved into a C# expression, and the argument type is stored for future reference.

The operation name and argument types make up the method's signature. The next step is to locate the corresponding method in the left side. Each method defined on the left side is examined. If a method contains the same name, same number of parameters, and same parameter types (in order), then a matching method has been located.

As OCL expressions must be guaranteed to be side effect free, the method must be marked with the "Query" attribute as stated in Section 5.2.7. The matching method is checked for the "Query" attribute. A compile-time error is generated if the attribute does not exist.

With the matching method located, and verified to be side effect free, the compiler completes the resolution of the operation expression by generating a C# expression that invokes the requested method using the given arguments. The result of the operation expression is the return value of the method, and the result type is the method's return type.

As stated in Chapter 2, C# provides for two parameter modifiers: reference (*ref*), and output (*out*). In the OCL, there is no notion of parameter modifiers. If a C# method matches an OCL operation call based on name and parameter types, the additional information specified by the parameter modifiers is not required. The OCL arguments will have the appropriate parameter modifiers added as needed so that the arguments can be used in the context of a C# method call, regardless of the required parameter modifiers.

Variable Expressions

As discussed in Chapter 3, variable expressions consist of anything that is defined via a name. In some cases, a variable expression is not necessarily a variable expression and requires a mapping to another expression type. The following sections will examine the resolution process for the various types of variable expressions.

Static Variables

A static variable expression consists of the following format: "<typename>::<variableName>". The resolution of a static expression begins with separating the type name from the variable name. Once the type name has been extracted, the type name is resolved. If a valid type is discovered, the resolution continues. If the beginning of the variable expression does not resolve into a type name, then the variable expression is not a static expression, and the expression is passed to the other types of variable expressions.

Once the type information has been derived, the compiler searches through the constants, fields, and properties to locate a member that matches the given variable name. If a corresponding member cannot

be located, a compile-time error is generated. Once the corresponding member has been located, the compiler will generate a C# member access expression to complete the resolution of a static variable expression.

Self Expressions

The OCL defines a keyword named *self*. The *self* keyword refers explicitly to the current contextual instance[45]. When the reference to the contextual instance is obvious, the use of the *self* keyword is optional. Figure 5.50 provides an example of the *self* keyword.

```
context Customer
inv: self.name = 'Dave'
```

Figure 5.50: Use of the *self* keyword

When the *self* keyword is encountered by the compiler, the compiler checks to see if there is a valid contextual instance. If such an instance cannot be located, as is the case with a static operation, a compile-time error is generated. If the contextual instance can be located, the compiler resolves the occurrence of the *self* keyword into the C# *this* keyword.

Result Expressions

In addition to the *self* keyword, the OCL also defines the *result* keyword. The *result* keyword can only be used in a postcondition. The *result* keyword is a reference to the return value of an operation. Figure 5.51 illustrates the use of the *result* keyword.

```
context Math::Square(i :Integer) : Integer
post: result = i * i
```

Figure 5.51: Use of the *result* keyword

The resolution process for the *result* keyword begins with a check to ensure that the keyword is being used in the context of a postcondition. If the keyword is not being used in a postcondition, a compile-time error is generated. The next step checks the operation to which the postcondition is being applied. If the method does not have a return value, a compile-time error is generated. The *result* keyword can

only be used on operations that return a value. The final step in resolving the *result* keyword is to generate a C# local variable reference to the result value. During the integration with a C# method or accessor, a postcondition is tested after the method has finished executing. If a method returns a value, a temporary local variable will be inserted to reference the result of the execution of the method's body. The details of the integration will be discussed later in this chapter.

Value Expressions

Value expressions are not part of the OCL specification. Value expressions are used to allow an OCL expression to reference the implicit C# value parameter. The implicit value parameter is used only in set accessors. The parameter indicates the new value of the property as shown in Figure 5.52.

```
class Customer
{
    private string _name;
    OCL
    [
        "context Customer::Name set: String"
        "pre: value.size() > 0"
    ]
    public string Name
    {
        set { _name = value; }
        get { return _name; }
    }
    public static void Main()
    {
        Customer c = new Customer();
        c.Name = Dave;
    }
}
```

Figure 5.52: Use of the value keyword

In order to allow expressions in the OCL to refer to the value parameter, a corresponding value expression has been devised. A value expression is defined by a variable expression using the name "value". When such an expression is discovered, the compiler checks to see if the value expression is being used in the context of a set accessor. If a value expression is being used outside of a set accessor, a compile-time error is not generated. An error is not generated because the user may have defined a parameter with the name "value".

Once the compiler has verified the value expression is being used in the context of a set accessor, the

compiler generates a C# parameter reference for the value parameter. There is no special code that must be generated in order to refer to a value parameter. The value parameter is accessed the same as an explicit parameter.

Let Variable Expressions

We have seen that a let expression can be used to define variables that refer to a complete OCL expression. In the body portion of the let expression, the variables are treated by the parser as variable expressions. In order to resolve the let variables, the compiler creates a cache of the variable names and values when it encounters a let expression. In the context of a variable expression, the variable name used may be in reference to a variable defined in a let expression.

In order to resolve a let variable, the compiler will extract the variable name from the variable expression and look in the cache. If the variable name is found in the cache, the compiler will issue C# code for a local variable reference. The local variable information is also contained in the cache. The variable itself is defined during the resolution of the let expression. If the variable name is not in the cache, the compiler continues with the next type of variable expression.

Iterator Variable Expressions

Iterator expressions also define iterator variables. These iterator variables are also inserted into a cache for use during the resolution of the iterator body expression. Once the body of the iterator expression has been resolved, the cache is cleared. If the compiler locates a valid variable name in the iterator cache, the variable is in scope and can be resolved. The resolution of iterator variable expressions is performed by a C# local variable reference.

Local Variable Expressions

The OCL is used to express constraints on structural elements of a system. The OCL is designed to be transformed via a CASE tool into the code level. In the case where the OCL is being implemented

at the behavioral level, rather than the structural level, the use of behavioral elements may be desired. In order to facilitate this, a provision for the referencing of local variables in postconditions has been implemented. In order to enable the use of local variable expressions in the OCL, the corresponding compiler option must be enabled. The option is not enabled by default.

When the use of local variable expressions is enabled, local variables can be referenced in postconditions only. If a local variable reference is used outside of a postcondition, the variable will not be resolved, and a compile-time error will be generated. The resolution of a local variable reference faces a major hurdle. Consider Figure 5.53.

```
class Math
{
    OCL
    [
        "context Math::Divide(a : Integer, b : Integer) : Real"
        "post: r = a / b"
    ]
    public double Divide(int a, int b)
    {
        if(b != 0)
        {
            double r = a / b;
            return r;
        }
        else
            throw new Exception("Divide By Zero");
    }
}
```

Figure 5.53: *Use of a local variable expression*

The figure creates a postcondition that ensures the local variable “r” contains the value of “a” divided by “b”. As postconditions are checked at the end of a method, the local variable “r” has gone out of scope, and will no longer be accessible outside of the *if* statement. In order to allow for variables defined in scopes that are not accessible by the postcondition to be used in a postcondition, the compiler will promote the variable to a valid scope. Figure 5.54 illustrates the result.

Variables are only promoted if they are referenced in the postcondition. Upon completion of the variable promotion process, the resolution is completed by creating a C# local variable reference.


```

class Math
{
    OCL
    [
        "context Math::Divide(a : Integer, b : Integer) : Real"
        "post: r = a / b"
    ]
    public double Divide(int a, int b)
    {
        double r = 0.0;
        if(b != 0)
        {
            r = a / b;
            return r;
        }
        else
            throw new Exception("Divide By Zero");
    }
}

```

Figure 5.54: *Promotion of a local variable for use in a local variable expression*

Parameter Expressions

If an OCL expression refers to a parameter, the resolution process is straightforward. The compiler checks the name of the variable expression against the parameter list. If there is a match, the compiler converts the variable expression into a C# parameter reference.

Field and Property Expressions

If a variable expression cannot be resolved by one of the previous methods, the next step is to see if the variable expression is referring to either a field or property defined on the current contextual instance. The resolution process begins by checking for fields and properties at the bottom level of the inheritance chain for a matching name. If a field or property has the same name as the variable expression, a C# member access expression is generated to perform the resolution.

If a suitable field or property cannot be found, the compiler searches through the inheritance hierarchy in a bottom-up fashion. If an accessible field or property is found in the inheritance hierarchy, the variable expression is resolved into a C# member access expression. If, after exhausting the inheritance hierarchy, the compiler cannot find a match for the variable expression, resolution continues to the final type of variable expressions: type expressions.

Type Expressions

The only remaining type of variable expression that exists is a type expression. A type expression occurs where the variable expression does not refer to a member or variable, but rather an actual type. Resolution of a type expression occurs by extracting the variable name from the variable expression and using the C# AST, as well as the list of imported types to locate a match. If a match is located, the compiler will create a C# type expression to resolve the variable expression. When a type expression is resolved, the type of the expression is not the type that is resolved, but rather an instance of the “System.Type” class. The “System.Type” class is used to represent all types in the system[26].

If after performing a type lookup on the variable expression a match cannot be found, the resolution concludes by the compiler issuing a compile-time error indicating that the name given in the variable expression cannot be located. With the resolution for variable expressions complete, the final type of expressions will be examined.

Literal Expressions

The majority of the literal expressions can be resolved during the constant resolution phase. The exception lies where a collection or tuple literal is created, and contains one or more elements that cannot be resolved into a constant, as in Figure 5.55.

```
Set {aPerson1, aPerson2}  
Tuple { a: String = aName, b: Integer = 25}
```

Figure 5.55: *Non-literal literal expressions*

The resolution of these types of literal expressions begins by resolving each of the elements using the resolution rules specified earlier in the chapter. Once each of the elements has been resolved, the types of the elements are verified. In the case of collections, each of the elements must have the same type. In the case of tuples, the element type must match the type provided in the tuple declaration. From Figure 5.55, the expression “aName” must resolve into the “String” type. In the case where the collection type is a “Set” or “OrderedSet”, the compiler must also ensure that no two expressions have the same value. If any of these checks fail, the compiler issues a compile-time error and the resolution fails.

Once each of the element expressions have been resolved and the types verified, the C# code generation phase begins. Unlike a collection or tuple that can be resolved into a constant, the collection or tuple is not instantiated at compile-time. Rather, the compiler will generate code that will allow for the collection or tuple to be instantiated at runtime. The examples in Figure 5.55 will be resolved as shown in Figure 5.56.

```
new OCL2STL.Set(new object[] {  
    <ResultOfaPerson1Resolution>,  
    <ResultOfaPerson2Resolution>});  
new OCL2STL.Tuple(new string[] {a, b},  
    new object[] { <ResultOfaNameResolution>, 25},  
    new object[] { new OCL2STL.String().GetType(),  
        new OCL2STL.Integer().GetType() }));
```

Figure 5.56: *Resolution of non-literal literal expressions*

5.4 Summary

The resolution process takes an OCL expression and converts it into a corresponding C# expression. An OCL expression is resolved in one of two ways: constant resolution or non-constant resolution. Constant resolution resolves the OCL expression into a constant value. The corresponding C# expression is a simple literal. Non-constant resolution of an OCL expression occurs when the constant resolution fails. The result of a non-constant resolution is a C# expression or group of expressions that yield the result of the OCL expression at runtime, as opposed to compile-time.

If each of the OCL expressions in a system is resolved without error, the integration process continues. The next step in the integration process is to integrate each of the C# expressions that were created to represent an OCL expression on to the main C# parse tree. The next Chapter will examine the integration process for each type of OCL expression.

Chapter 6

Compilation

With the resolution of each OCL expression complete. The next step is to integrate each of the resolved OCL expressions with their corresponding C# element. This chapter will provide details of the integration process. The integration process involves taking each individual parse tree that represents an OCL expression and grafting it to the main C# parse tree. The chapter finishes with a look at the actual code generation step, some of the optimizations used, and information how the compiler was tested.

6.1 Mapping Init Rules to C# Elements

When an initialization rule is encountered, the rule states that the starting value for a field or constant is being specified. The expression that represents the value to be assigned to the field is resolved using the constant resolution rules. If the expression cannot be resolved into a constant, a compile-time error is generated. An initialization rule expression must result in a constant. The constant value returned by the resolution phase will be used to initialize the field or constant to which the initialization rule is assigned.

6.1.1 Fields

After the constant value is resolved, the constant's type is compared with the field's type. If the constant value cannot be converted into the field's type, a compile-time error is generated. Once the field type and the constant's type have been verified compatible, the mapping of the init rule is completed by modifying the field to include an initial value.

In the compiler, a field is represented by the "Field" class. The "Field" class contains a property to indicate if the field contains an initial value or not. The constant specified by the initialization rule is used as the initial value. During code generation, the field's initial value is inserted into each of the constructors to assign the initial value. Figure 6.1 shows the resulting IL code that initializes a field named "_f1" of type "Byte" a value of "42". Unlike C++, C# does not implement the initialization of fields directly. Rather, C# initializes a field by adding an assignment statement at the beginning of the constructor(s) defined on the corresponding container. If such a constructor does not explicitly exist, one will be created. This means the IL definition of a field does not specify an initial value for the field. Figure 6.2 illustrates the definition of the "_f1" field.

```
ldarg.0
ldc.i4.s 42
stfld unsigned int8 DaveArnold.MyClass::_f1
```

Figure 6.1: *IL code to initialize a field*

6.1.2 Constants

As with fields, constant values can also be defined in the OCL by using initialization rules; after the constant value is resolved, and the constant's type is compared with the C# constant's type. As all C# constants must be defined with a value, the value specified by the OCL initialization rule will be in addition to the value already specified in C#. Figure 6.3 illustrates this condition. The current C# AST has already fully resolved the constant, and the value specified at the C# level.

```
.field private unsigned int8 _f1
```

Figure 6.2: *Definition of a field named _f1*

```
class MyClass
{
    OCL
    [
        "context MyClass::_cl : String"
        "init: 'Bye World'"
    ]
    private const string _cl = Hello World;
}
```

Figure 6.3: *Definition of a constant in C#*

The value specified by the OCL initialization rule will be used in place of the value specified in C#. As the two values may not be equal, a compile-time warning is generated to indicate that the value specified in C# will be replaced with the OCL value.

C# implements constants differently than fields. As previously stated, fields are initialized separately from their definition. Conversely, constants are initialized and defined at the same time. Figure 6.4 contains the resultant code for the constant specified in Figure 6.3.

```
.field private literal string _cl = "Bye World"
```

Figure 6.4: *Definition of a constant named _cl*

6.1.3 Summary

Initialization rules can apply to either fields or constants. An initialization rule specifies either the initial value for a field, or the value of a constant. Fields are initialized via assignment statements at the beginning of all constructors. Constants are initialized during their definition. When constants are initialized by an OCL expression, the constant initialization value specified in C# will be replaced with the OCL value.

6.2 Mapping Derive Rules to C# Elements

OCL derive rules allow a field to be defined in terms of an OCL expression. During the assignment of the derive rule to the C# field, the compiler created an empty read-only property to represent the field.

The remaining task is to resolve the OCL expression and use the C# code generated by the resolution to complete the body of the get accessor defined in the property.

The OCL expression is resolved using the resolution rules discussed in Chapter 5. Unlike initialization rules, the OCL expression used in the derive rule does not have to be resolved into a constant. Following successful resolution of the OCL expression, the type of the expression is checked against the type of the property. In the case where the two types do not match, a compile-time error is generated. The types will not match if the resultant type of the OCL expression, and the type defined in the OCL derive rule, are not the same.

With the expression resolved, and the types checked, the compiler completes the mapping of the derive rule by using the resolved OCL expression to compute the resultant value of the property. The value is returned to the caller via the use of a C# *return* statement. Figure 6.5 shows the use of an OCL derive rule, while Figure 6.6 shows the generated code. The first portion of Figure 6.6 shows the definition of the “_d” property. The property is defined to be read-only, as it only contains a get accessor. The get accessor is implemented via the “get__d()” method.

```
class MyClass
{
    OCL
    [
        "context MyClass::_d : Integer"
        "derive: Tuple{ name = 'Dave', age = _age}.age * 2"
    ]
    private int _age = 25;
}
```

Figure 6.5: A derive expression for the _d field

```
.property specialname int32 _d()
{
    .get instance int32 MyClass::get__d()
}
.method public hidebysig specialname instance int32
    get__d() cil managed
{
    .maxstack 5
    .locals init (string[] V_0, object[] V_1, object[] V_2)
    IL_0000: ldc.i4.2
    IL_0001: newarr [mscorlib]System.String
    IL_0006: stloc.0
    IL_0007: ldloc.0
```

```

IL_0008: ldc.i4.0
IL_0009: ldstr "name"
IL_000e: stelem.ref
IL_000f: ldloc.0
IL_0010: ldc.i4.1
IL_0011: ldstr "age"
IL_0016: stelem.ref
IL_0017: ldloc.0
IL_0018: ldc.i4.2
IL_0019: newarr [mscorlib]System.Object
IL_001e: stloc.1
IL_001f: ldloc.1
IL_0020: ldc.i4.0
IL_0021: ldstr "Dave"
IL_0026: stelem.ref
IL_0027: ldloc.1
IL_0028: ldc.i4.1
IL_0029: ldarg.0
IL_002a: ldfld int32 MyClass::_age
IL_002f: box [mscorlib]System.Int32
IL_0034: stelem.ref
IL_0035: ldloc.1
IL_0036: ldc.i4.2
IL_0037: newarr [mscorlib]System.Object
IL_003c: stloc.2
IL_003d: ldloc.2
IL_003e: ldc.i4.0
IL_003f: ldstr "Dave"
IL_0044: stelem.ref
IL_0045: ldloc.2
IL_0046: ldc.i4.1
IL_0047: ldarg.0
IL_0048: ldfld int32 MyClass::_age
IL_004d: box [mscorlib]System.Int32
IL_0052: stelem.ref
IL_0053: ldloc.2
IL_0054: newobj instance void
    [OCL2STL]DaveArnold.OCL2STL.BaseTypes.Tuple::.ctor(
        string[], object[], object[])
IL_0059: ldstr "age"
IL_005e: callvirt instance object
    [OCL2STL]DaveArnold.OCL2STL.BaseTypes.Tuple::access(string)
IL_0063: unbox [mscorlib]System.Int32
IL_0068: ldind.i4
IL_0069: ldc.i4.2
IL_006a: mul.ovf
IL_006b: ret
}

```

Figure 6.6: *IL code for a derive rule*

The majority of the “get__d()” method is used to initialize the tuple literal. The tuple must be created at runtime because the value of the “age” element cannot be determined under constant resolution. Once the tuple has been created, the “access” method defined on the “Tuple” class is used to access the value

of the “age” element. That value is then multiplied by “2”, and the result returned to the caller.

6.3 Mapping Def Rules to C# Elements

Definition rules are used to define named attributes and operations. In the case of an attribute, the compiler created a C# read-only property with the name used in the definition rule. In the case of an operation, the compiler has also created a corresponding operation with the name, return type, and parameters that were specified in the definition rule.

The remaining task is to complete the implementation of the get accessor in the case of an attribute, or the method in the case of an operation. The first step in completing the implementation is to resolve the OCL expression that makes up the defined attribute or operation. Once the OCL expression is successfully resolved, the expression type is verified against the property or method return type. If there is a discrepancy, a compile-time error is generated.

Once the OCL expression has been resolved and its type verified, the compiler uses a *return* statement to return the value of the statement to the caller. In the case of a property, the code generated is the same as shown in Figure 6.6. In the case of a method, the code generated is the same, with the exception that it is not placed in a get accessor, but rather the code is used directly as the method’s body.

6.4 Mapping Body Rules to C# Elements

A body rule is used to define an implementation for a method. The compiler has already created a new method with no implementation to represent the body rule. The remaining task is to take the OCL expression used to represent the body expression and use it to implement the method’s body.

The OCL expression is first resolved using the resolution rules presented in Chapter 5. Once a successful resolution has occurred, the compiler will check the result type of the resolution with the return type of the method. If the two types do not match, a compile-time error is issued. Once the types have been

verified, the resolved OCL expression is used to generate the value for the method to return. That value is returned via a C# *return* statement. Figure 6.7 provides an example body statement. Where Figure 6.8 shows the resultant code generated by the compiler.

```

OCL
[
  "context MyClass::Multiply(a : Integer, b : Integer) : Integer"
  "body: a * b"
]

```

Figure 6.7: *Body expression for the multiplication of two numbers*

```

.method public instance class
  [OCL2STL]DaveArnold.OCL2STL.PrimitiveTypes.Integer
  Multiply(
    class [OCL2STL]DaveArnold.OCL2STL.PrimitiveTypes.Integer a,
    class [OCL2STL]DaveArnold.OCL2STL.PrimitiveTypes.Integer b)
  cil managed
  {
    .custom instance void
      [OCL2STL]DaveArnold.OCL2STL.QueryAttribute::.ctor() =
        ( 01 00 00 00 )
    .maxstack 2
    IL_0000: ldarg.1
    IL_0001: ldarg.2
    IL_0002: call class
      [OCL2STL]DaveArnold.OCL2STL.PrimitiveTypes.Integer
      [OCL2STL]DaveArnold.OCL2STL.PrimitiveTypes.Integer::
        op_Multiply(
          class [OCL2STL]DaveArnold.OCL2STL.PrimitiveTypes.Integer,
          class [OCL2STL]DaveArnold.OCL2STL.PrimitiveTypes.Integer)
    IL_0007: ret
  }

```

Figure 6.8: *IL code for a body expression*

The code listing in Figure 6.8 contains an application of the “Query” attribute. The “Query” attribute is used to indicate that the “Multiply” operation is side effect free. As previously stated, any operation defined by an OCL expression must be side effect free, as all OCL expressions are side effect free. Following the application of the “Query” attribute, the body of the method loads the two parameters onto the stack, and then uses the multiply operation defined on the OCL “Integer” type to compute the result. That result is then returned to the caller.

6.5 Mapping Constraints to C# Elements

The final types of OCL expressions to be mapped are preconditions, postconditions, and class invariants. Preconditions and postconditions can be expressed on any method or accessor. Invariants are specified at the class level. The invariants can be seen as being applied to each method and accessor in the class. The constraints are mapped after all of the other OCL rules are mapped to their respective C# elements. The constraints must be mapped last because some of the previous OCL expressions result in the creation of a new method or accessor. The new methods and accessors will need to have the class invariants applied. The following sections will illustrate the mapping process for a single method. The process is followed for each method and accessor defined in the system.

6.5.1 Resolution of Constraints

Before the mapping can begin, each precondition, postcondition, and class invariant expression is resolved, using the resolution rules previously discussed. Once each constraint is resolved it is assigned to the corresponding method or accessor for later use. Each of the constraints must resolve into an expression of the “Boolean” type. The result of the evaluation of a constraint is either success or failure.

The resolution of a class invariant must be performed for each method or accessor. If the class invariant was only resolved once for the entire class, there would be a problem if the class invariant contained an expression that required extra variable space. Each instance of extra variable space must be defined on the method or accessor where the space is required. In order to accommodate this issue, if a class invariant is defined on a class that contains five methods, the class invariant is resolved five separate times; once for each method. The reader should note that class invariants are tested at the beginning and at the end of a method or accessor. The invariant used in both locations is only resolved once, as the extra variable space can be shared because the invariants are being used in the context of the same method or accessor.

If a class implements one or more interfaces during the constraint resolution process, any constraints that exist at the interface level are resolved and assigned to the corresponding method or accessor defined

in the class. The constraints are flattened and all resolved in the context of the implementation class. Each constraint's source is preserved so that during the implementation of Findler's method, blame can be assigned to the correct party.

If a class inherits from another class, a similar method is used to resolve each of the base class' constraints at the subclass level. The subclass has a copy of all the corresponding constraints, as well as sufficient source information so that the code generator can determine the hierarchy level where the constraint is defined.

Upon the successful resolution of each of the constraints, and their assignment to the corresponding method or accessor, the C# code generation process begins.

6.5.2 Value Saving

The first step in the code generation process is to preserve any values that are referenced in a postcondition using the *@pre* modifier. The *@pre* modifier indicates that the value of an attribute or operation before the execution of the method is requested, opposed to the current value. In order to facilitate the *@pre* modifier, a temporary local variable is used to store the value of the attribute or operation as previously mentioned.

The code generation process consists of creating the local variable. The variables type is determined by either the attribute's type or the return type of the operation. Once the variable has been defined, an assignment statement is used to assign the value of the attribute or operation to the local variable. The local variable now holds the value of the attribute or operation before the body of the method executes.

With the value of the attribute or operation saved, the postcondition will refer to the local variable rather than the attribute or operation. The resolution of the postcondition will resolve the use of the *@pre* modifier into the corresponding local variable.

In addition to the local variables introduced by the use of the *@pre* modifier, OCL let expressions can also introduce additional local variables. If a let expression is used in the context of a precondition or a class invariant, a variable is used to reference the value of a larger OCL expression. The variables defined

under the OCL are mapped into a corresponding C# local variable. The variable's type is dependent on the type specified in the OCL expression. Once the variable is initialized, an assignment statement is used to assign the variable the result of evaluating the larger OCL expression.

As stated in Section 5.3.2, the compiler uses a map to replace references to the let variable defined in the OCL, with the local variable defined in C#. In the implementation, the let variables are assigned before any other activity. The let variables are assigned first because they may be referenced in any other OCL expression, including expressions that contain the *@pre* modifier.

In the OCL specification, let variables can only be used within the scope that they are defined in[32]. However our code generation phase does not pay respect to the scope of let variables. In terms of a method, the C# representation of a let variable is available throughout the method. The enforcement of a let variable's scope is done during the expression resolution phase. If a let variable is referenced outside of its scope, a compile-time error is generated. With the requested values saved for later use, the code generation for OCL constraints begins.

6.5.3 Class Invariants - Part 1

The second step in the code generation process is to generate code for checking the class invariants. Class invariants are checked at both the beginning and the end of the method. At this stage, only the code for the invariants at the beginning of the method is generated. There are two cases where class invariants are not checked at the beginning of a method. Class invariants are not checked if the method contains the *static* modifier, as the method resides at the class level as opposed to the object level. The second and more interesting case, is where class invariants are not checked at the beginning of constructors. By not checking class invariants until the end of constructors, the constructor is able to initialize the object's state. If either of these two cases is present, the compiler skips the generation of the invariant code and jumps down to the generation of precondition code.

If neither of the two cases applies, generation of code to check the class invariants begins. The code generation process begins by introducing a new local variable of type "Boolean" to store the result of testing each class invariant. Once a local variable has been added for each invariant, an assignment

statement for each invariant is added immediately following the declaration of the new variables. The assignment statement results in the variable being assigned a value of true if the invariant is valid or false otherwise.

Once each of the local variables are assigned, an *if* statement is generated to test the values of the local variables to see if any of the invariants have failed. If all of the invariants are valid, then execution continues. When an invariant fails, an exception is raised. If there is more than one invariant, sub *if* statements are used to determine which invariant failed. If more than one invariant fails, only the first one discovered will be reported. The compiler will generate exception-raising code based on the type of exception specified by the compile-time options. Chapter 5 examined exception types.

In terms of inheritance, all invariants defined at any level of the inheritance hierarchy are flattened and processed as if they were defined on the method's containing class. The source of the invariant is preserved, so the class on which the invariant is defined can be referenced during the raising of an exception.

Once code has been generated to evaluate, test, and generate exceptions for each class invariant, generation of precondition code follows.

6.5.4 Preconditions

The third step in the code generation process is the generation of code for checking preconditions. The process starts by working with the locally defined preconditions. Preconditions that are defined on interfaces or superclasses are not examined. As preconditions may consist of complex statements, each precondition will be assigned to a variable of type "Boolean". Thus, the first phase of generating code for preconditions is to define a new variable for each of the preconditions. Once a variable has been defined, an assignment expression is prepended to the method's body that assigns the result of executing the precondition to the new boolean variable.

The next phase in the code generation process is to insert an *if* statement following the boolean variable assignment statements. The *if* statement checks to see if any of the boolean variables holds a false value.

A false value means that one of the preconditions has failed. The process begins by only working with the preconditions that were defined at the current level of the hierarchy. If a precondition fails, the blame is assigned to the caller of the method.

The body of the *if* statement contains code that will respond to the failure of a precondition. As stated in Chapter 5, the exception raising code will consist of the generation of a C# exception, the displaying of a dialog box, or both. If there is more than one precondition, then the body of the *if* statement will consist of sub *if* statements to determine which precondition failed. If more than one precondition failed, only the first failing precondition is reported.

If the method being processed directly implements an interface, then preconditions defined on the interface are also included in the code generation process. The difference is that the main *if* statement is replaced with separate *if* statements for the local preconditions, and the preconditions specified at the interface level. If both the local and the interface preconditions evaluate to true, then there is no failure, and execution of the method continues. If both of the local and the interface preconditions fail, then the caller is blamed in the resulting exception raising code. If the preconditions at the interface level are valid, but the local preconditions fail, the inheritance hierarchy is invalid. In this case, the exception raising code places blame on the method's containing class. In the reverse case, where the local preconditions are valid, but the interface level preconditions fail, the inheritance hierarchy is valid, and the execution of the method continues. Figure 6.9 provides the truth table for the processing of preconditions.

C - true, I - true:	No exception, execution continues
C - false, I - false:	Exception raised, the caller is to blame
C - false, I - true:	Exception raised, C is to blame
C - true, I - false:	No exception, execution continues

Figure 6.9: Truth table for processing preconditions

In the final case where the local preconditions are valid, but the interface preconditions are not, there should be a violation if the caller of the method is using the object as an instance of the interface, rather than an instance of the class. After examining various methods of implementation, we remark that determining the context of the object used by the caller at runtime is non-trivial. Thus, this last case is not implemented. Implementation could be achieved if the method could check the instance for which

the method is called.

The remaining case for mapping preconditions occurs when the inheritance hierarchy consists of more than just an interface. A recursive method is used to generate C# code for testing and assigning blame to the respective offender. Before code is generated for the superclass, the compiler checks to see if the method contains the *new* modifier. If the *new* modifier is present, preconditions defined further up the inheritance hierarchy are not tested. The rationale for this was presented in Chapter 4. If the *new* modifier is not present, the compiler works up the inheritance chain, generating code for testing each level of preconditions, generating code to test, and then raise exceptions for each precondition defined in the inheritance chain. Figure 6.10 illustrates such an inheritance chain, and Figure 6.11 illustrates the resultant code.

```

interface ITest
{
    OCL
    [
        "context ITest::Square(i : Integer) : Integer"
        "pre: i > 3"
    ]
    int Square(int i);
}
class Squares : ITest
{
    OCL
    [
        "context Squares::Square(i : Integer) : Integer"
        "pre: i > 2"
    ]
    public virtual int Square(int i) { return i*i; }
}
class BetterSquares : Squares
{
    OCL
    [
        "context BetterSquares::Square(i : Integer) : Integer"
        "pre: i > 1"
    ]
    public override int Square(int i)
    {
        return i*i;
    }
}

```

Figure 6.10: Precondition inheritance chain


```

.method public hidebysig virtual instance int32
Square(int32 i) cil managed
{
    .maxstack 20
    .locals init (
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_0,
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_1,
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_2)
    IL_0000: ldarg.1
    IL_0001: ldc.i4.2
    IL_0002: cgt
    IL_0004: call class
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Explicit(bool)
    IL_0009: stloc.1
    IL_000a: ldarg.1
    IL_000b: ldc.i4.3
    IL_000c: cgt
    IL_000e: call class
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Explicit(bool)
    IL_0013: stloc.2
    IL_0014: ldarg.1
    IL_0015: ldc.i4.1
    IL_0016: cgt
    IL_0018: call class
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Explicit(bool)
    IL_001d: stloc.0
    IL_001e: ldloc.0
    IL_001f: call bool
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Explicit(
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
    IL_0024: brtrue IL_0048
    IL_0029: ldstr "Pre-Condition Failed."
    IL_002e: ldstr "Client"
    IL_0033: ldstr "context BetterSquares::Square(i : Integer) : In"
        + "teger\r\npre: i > 1"
    IL_0038: ldstr "'BetterSquares::Square'"
    IL_003d: ldstr "Pre-Condition"
    IL_0042: newobj instance void
        [OCL2STL]OCL2STL.OCLException::.ctor(string,
        string, string, string)
    IL_0047: throw
    IL_0048: ldarg.1
    IL_0049: ldarg.1
    IL_004a: mul.ovf
    IL_004b: ret
    IL_004c: ldloc.2
    IL_004d: call bool
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Explicit(
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
    IL_0052: brtrue IL_0076
    IL_0057: ldstr "Pre-Condition Failed."
    IL_005c: ldstr "Squares"
    IL_0061: ldstr "context ITest::Square(i : Integer) :
        Integer\r\npre:"
        + " i > 3"
    IL_0066: ldstr "'BetterSquares::Square'"

```

```

IL_006b: ldstr "Pre-Condition"
IL_0070: newobj instance void
        [OCL2STL]OCL2STL.OCLException::.ctor(string,
        string, string, string, string)
IL_0075: throw
IL_0076: ldloc.1
IL_0077: call bool
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
IL_007c: brtrue IL_00a0
IL_0081: ldstr "Pre-Condition Failed."
IL_0086: ldstr "BetterSquares"
IL_008b: ldstr "context Squares::Square(i : Integer) :
        Integer\r\npr"
        + "e: i > 2"
IL_0090: ldstr "`BetterSquares::Square'"
IL_0095: ldstr "Pre-Condition"
IL_009a: newobj instance void
        [OCL2STL]OCL2STL.OCLException::.ctor(string,
        string, string, string, string)
IL_009f: throw
}

```

Figure 6.11: *IL code to implement a precondition inheritance chain*

The code listing in Figure 6.11 begins by defining three local variables of the “Boolean” type to store the results of evaluating the preconditions. The code then evaluates each of the three preconditions, and stores the result of the precondition in one of the local variables. Once the assignment is complete, several tests are performed to determine if any of the precondition expressions have yielded a false value. If a false value is found, code to raise an exception is executed. There are three blocks of code to raise exceptions. Each block assigns blame to the correct element, depending on which precondition failed.

6.5.5 Method Path Modification

If a method only has preconditions defined, then the code generation process is complete, and no further changes to the method are required. If the method contains postconditions, or is contained by a class that has class invariants, further processing is required. In the case of postconditions and class invariants, additional code must be generated upon completion of the method’s body.

A method body may contain several execution paths. Following the execution of each path, postconditions and class invariants must be checked. In order to reduce code bloat, each method’s execution paths

are merged into a single path, allowing the postcondition and class invariant code to be generated only once.

The action of merging a method's execution paths consists of locating each of the *return* statements. Once a *return* statement is located, it is replaced with a *goto* statement, which jumps to the end of the method's body. At the end of the method's body, at the jump location, the postcondition and class invariant code is inserted. In the case where a value is returned, a temporary variable is used to store the return value, and upon successful evaluation of the postconditions and class invariants, the value stored in the temporary variable is returned. Figure 6.12 illustrates an example method body before merging the method's execution path. Figure 6.13 illustrates the resultant method.

```
public int Abs(int x)
{
    if(x < 0)
        return x;
    return x;
}
```

Figure 6.12: *Method body before merging execution paths*

```
public int Abs(int x)
{
    int ocl_result_value = 0;
    if(x < 0)
    {
        ocl_result_value = -x;
        goto ocl_result_check;
    }
    ocl_result_value = x;
    goto ocl_result_check;
ocl_result_check:
    // Postconditon and class invariant code
    return ocl_result_value;
}
```

Figure 6.13: *Method body after merging execution paths*

In Figure 6.13, the method's body starts with the definition of a local variable to store the return value. The variable's type is the same as the method's return type. Each of the *return* statements have been replaced with an assignment to the local variable, and then a *goto* statement to jump to the end of the method's body. At the jump target, code to check the postconditions and class invariants is inserted. Finally, the return value stored in the local variable is returned to the caller. With the method's execution paths merged into one, the next step is to generate code to check the validity of the postconditions.

6.5.6 Postconditions

The fifth step is the generation of code to check the postconditions assigned to a method. The code generation process for postconditions is analogous to the code generation process for preconditions. The process begins by creating a local variable to store the result of executing each of the postconditions. As previously stated, each postcondition must evaluate into either a true or a false value. Thus, each of the new local variables is of the “Boolean” type. Immediately following the definition of the local variables, an assignment statement is used to assign the value of each postcondition to a corresponding variable.

Once the results of evaluating the postconditions have been assigned to a boolean variable, an *if* statement is used to determine if any of the boolean variables contains a false value. A false value indicates that a postcondition has failed. As with preconditions, the body of the *if* statement contains code that will respond to the failure of a postcondition. If there is more than one postcondition, then the body of the *if* statement will consist of sub *if* statements to determine which postcondition failed. If more than one postcondition failed, only the first one will be reported.

The use of postconditions with respect to inheritance follows the same code generation process as preconditions. The only difference is the truth table used to assign blame is the one shown in Figure 6.14, as opposed to the one used for preconditions (Figure 6.9).

```
C - true, I - true:   No exception, execution continues
C - false, I - false: Exception raised, the method is to blame
C - false, I - true:  Exception raised, the method is to blame
C - true, I - false:  Exception raised, C is to blame
```

Figure 6.14: *Truth table for processing postconditions*

In order to visualize how inheritance works with postconditions, consider the code listing in Figure 6.15. The resultant code that is generated by the code generator is displayed in Figure 6.16. Once the postcondition code generation process has been completed, the final generation step begins, namely, Class Invariants - Part 2.

```

interface ITest
{
    OCL
    [
        "context ITest::Square(i : Integer) : Integer"
        "post: result = i * i"
    ]
    int Square(int i);
}
class Squares : ITest
{
    OCL
    [
        "context Squares::Square(i : Integer) : Integer"
        "post: result = i@pre * i@pre"
    ]
    public virtual int Square(int i) { return i*i; }
}
class BetterSquares : Squares
{
    OCL
    [
        "context BetterSquares::Square(i : Integer) : Integer"
        "post: result = i * i"
    ]
    public override int Square(int i)
    {
        return i*i;
    }
}

```

Figure 6.15: Postcondition inheritance chain

```

.method public hidebysig virtual instance int32
    Square(int32 i) cil managed
{
    .maxstack 33
    .locals init (
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_0,
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_1,
        int32 V_2,
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_3,
        int32 V_4,
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_5,
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean V_6)
    IL_0000: ldc.i4.0
    IL_0001: stloc.s V_4
    IL_0003: ldarg.1
    IL_0004: stloc.2
    IL_0005: ldarg.1
    IL_0006: ldarg.1
    IL_0007: mul.ovf
    IL_0008: stloc.s V_4
    IL_000a: br IL_000f
    IL_000f: ldloc.s V_4
    IL_0011: ldarg.1
    IL_0012: ldarg.1

```

```

IL_0013: mul.ovf
IL_0014: ceq
IL_0016: call class
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(bool)
IL_001b: stloc.1
IL_001c: ldloc.1
IL_001d: call bool
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(
class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
IL_0022: brtrue IL_00b8
IL_0027: ldloc.s V_4
IL_0029: ldloc.2
IL_002a: ldloc.2
IL_002b: mul.ovf
IL_002c: ceq
IL_002e: call class
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(bool)
IL_0033: stloc.s V_6
IL_0035: ldloc.s V_4
IL_0037: ldarg.1
IL_0038: ldarg.1
IL_0039: mul.ovf
IL_003a: ceq
IL_003c: call class
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(bool)
IL_0041: stloc.s V_5
IL_0043: ldloc.s V_6
IL_0045: call bool
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(
class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
IL_004a: brfalse IL_006e
IL_004f: ldstr "Post-Condition Failed."
IL_0054: ldstr "BetterSquares"
IL_0059: ldstr "context ITest::Square(i : Integer) :
    Integer\r\npost"
    + ": result = i * i"
IL_005e: ldstr "' BetterSquares::Square'"
IL_0063: ldstr "Post-Condition"
IL_0068: newobj instance void
    [OCL2STL]OCL2STL.OCLException::.ctor(string,
    string, string, string, string)
IL_006d: throw
IL_006e: ldloc.s V_5
IL_0070: call bool
    [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(
class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
IL_0075: brfalse IL_0099
IL_007a: ldstr "Post-Condition Failed."
IL_007f: ldstr "BetterSquares"
IL_0084: ldstr "context ITest::Square(i : Integer) :
    Integer\r\npost"
    + ": result = i * i"
IL_0089: ldstr "' BetterSquares::Square'"
IL_008e: ldstr "Post-Condition"
IL_0093: newobj instance void
    [OCL2STL]OCL2STL.OCLException::.ctor(string,

```

```

        string, string, string, string)
IL_0098: throw
IL_0099: ldstr "Post-Condition Failed."
IL_009e: ldstr "Supplier"
IL_00a3: ldstr "context BetterSquares::Square(i : Integer) : In"
        + "teger\r\npost: result = i * i"
IL_00a8: ldstr "'BetterSquares::Square'"
IL_00ad: ldstr "Post-Condition"
IL_00b2: newobj instance void
        [OCL2STL]OCL2STL.OCLException::.ctor(string,
        string, string, string, string)
IL_00b7: throw
IL_00b8: ldloc.s V_4
IL_00ba: ldloc.2
IL_00bb: ldloc.2
IL_00bc: mul.ovf
IL_00bd: ceq
IL_00bf: call class
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(bool)
IL_00c4: stloc.0
IL_00c5: ldloc.s V_4
IL_00c7: ldarg.1
IL_00c8: ldarg.1
IL_00c9: mul.ovf
IL_00ca: ceq
IL_00cc: call class
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(bool)
IL_00d1: stloc.3
IL_00d2: ldloc.0
IL_00d3: call bool
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
IL_00d8: brtrue IL_00fc
IL_00dd: ldstr "Post-Condition Failed."
IL_00e2: ldstr "Supplier"
IL_00e7: ldstr "context Squares::Square(i : Integer) :
        Integer\r\npo"
        + "st: result = i@pre * i@pre"
IL_00ec: ldstr "'BetterSquares::Square'"
IL_00f1: ldstr "Post-Condition"
IL_00f6: newobj instance void
        [OCL2STL]OCL2STL.OCLException::.ctor(string,
        string, string, string, string)
IL_00fb: throw
IL_00fc: ldloc.3
IL_00fd: call bool
        [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean::op_Implicit(
        class [OCL2STL]OCL2STL.PrimitiveTypes.OclBoolean)
IL_0102: brtrue IL_0126
IL_0107: ldstr "Post-Condition Failed."
IL_010c: ldstr "BetterSquares"
IL_0111: ldstr "context ITest::Square(i : Integer) :
        Integer\r\npost"
        + ": result = i * i"
IL_0116: ldstr "'BetterSquares::Square'"
IL_011b: ldstr "Post-Condition"
IL_0120: newobj instance void

```

```
        [OCL2STL]OCL2STL.OCLException::.ctor(string,  
            string, string, string, string)  
IL_0125: throw  
IL_0126: ldloc.s V_4  
IL_0128: ret  
}
```

Figure 6.16: *IL code to implement a postcondition inheritance chain*

6.5.7 Class Invariants - Part 2

The final step in the code generation process is the generation of code to validate the class invariants at the end of method execution. The code generation process is the same as the code generation process used to validate the class invariants at the beginning of the method, with one exception: invariant code is generated for constructors. In the ending section, invariant code is generated for constructors, but not destructors. By not validating invariants in a destructor, the destructor can clean up any resources used, without having to worry about satisfying the invariants. However, the invariants are validated at the beginning of destructors.

6.5.8 Summary

The preceding six steps are applied to each method and accessor in the system. During the execution of the six steps, code is inserted into each method and accessor to validate preconditions, postconditions, and class invariants. If during the evaluation of a constraint, it is found that the constraint does not hold, an exception will be raised. The mapping of the OCL constraints to C# marks the completion of the OCL compilation process. For more information on the semantic pass, and the generation of code for the OCL elements, please see the implementation[2]. The implementation of the OCL expression resolution and integration with the main AST required approximately forty thousand lines of code.

6.6 Code Generation

The result of the C# code generation for each of the OCL elements outlined in the preceding sections is a single AST that contains C# elements. These are a direct result of the resolution of the OCL expressions. The code that was generated from the various OCL elements is inserted directly into the AST, resulting in a single AST that can be used for binary code generation.

The only remaining task is to perform the binary code generation. In reality, directly executable code is not generated, but rather IL code. As discussed in Chapter 2, IL code is used as byte-code. The byte-code is then fed into the JIT compiler that produces an executable image.

As all of the OCL expressions have been resolved and mapped into the C# AST, using C# elements, the code generation process consists of restarting the Mono C# compiler. The compiler will then traverse through the AST and for each node in the tree, generate code via the “System.Reflection.Emit” API. The “System.Reflection.Emit” API provides a set of classes that are used to generate IL code[26].

6.7 Optimization

The Mono C# compiler contains several optimizations at both the parser level, and the code generation level. As our compiler uses the parser and the code generator provided by the Mono C# compiler, our compiler also takes advantage of these optimizations.

In terms of the OCL, the major optimization was already discussed in Chapter 5, with the resolution of constant expressions. The compiler will instantiate and evaluate any expression that can be fully resolved at compile-time. By using the result of the evaluation rather than the actual expression, run-time evaluation time can be reduced.

In terms of the compilation process, the action of performing a type lookup can be time consuming, especially in a system that contains thousands of types. In order to reduce the time required to perform a type lookup, several caches are used to reduce the time needed to lookup frequently requested types.

We are certain that several other optimizations could be implemented, especially during the resolution of non-constant OCL expressions. As the compiler was designed in the context of an academic exercise, these optimizations are reserved for future work.

6.8 Overall Compiler Flow: The Big Picture

This chapter, along with Chapters 3 and 4, have outlined a complete process for integrating OCL expressions with the C# language. This section will provide a brief summary of the entire compilation process. The process is illustrated in Appendix B. The blue boxes represent elements already present in the Mono compiler; the green boxes represent elements that were constructed to support the OCL. The error manager is a shared component, and has been modified to support OCL error messages.

The compilation process begins by splitting the source text into two groups: C# and OCL. Each source text element is passed through its respective lexical analyzer (tokenizer), parser and semantic analyzer. The C# source text is processed first so that following the completion of the semantic analysis, a reference model, can be generated. The reference model is used to represent the types that were either defined or imported in the C# source text. The reference model is used by the OCL semantic analyzer to lookup types that are referenced in the OCL.

Once the reference model is constructed, each of the OCL source text blocks are passed through the OCL lexical analyzer, parser, and semantic analyzer. Once the OCL semantic analysis is completed, each of the resolved OCL elements are merged into the C# AST (green triangle). Finally, once a single AST has been created, the C# code generator is used to complete the compilation process.

The entire implementation of our compiler is comprised of three hundred and eleven files contributing over two hundred and thirty thousand lines of C# source code. The source code line distribution is shown in Table 6.1.

Item	Lines
Original C# compiler	46,732
OCL and C# compiler	101,905
The OCL type library	6,687
Integrated development environment	74,759
Miscellaneous	12,286

Table 6.1: *Source code line distribution*

6.9 Validation

Upon completion of the implementation, a test suite was developed to validate the compiler. The test suite consists of thirty C# files augmented with various OCL expressions. The files consist of independent tests for every type of OCL expression discussed in this chapter.

In addition to the validation of OCL expressions, there are also a series of tests that were used to validate the compiler. For each error and warning message located in Appendix E, two tests were devised. The first test consists of an OCL expression designed to explicitly raise the given error or warning. The second test uses a similar OCL expression to successfully test the given error or warning. For example, consider the OCL expression in Figure 6.17.

```

OCL
[
  "context CollectionTests::_c1 : Integer"
  "init: Set{1, 2, 3, 4, 4}.size()"
]

```

Figure 6.17: *Test case to generate error OCL0085*

The expression states that the attribute “_c1” is to have an initial value equal to the size of the provided set literal. As the “Set” type is not allowed to contain duplicate elements, the compiler will issue the following error: “error OCL0085: Sets and OrderedSets cannot contain duplicate elements”. The OCL expression for the second test would be exactly the same as the one in Figure 6.17, except the second “4” would be removed.

The result of these tests ensures that the compiler is able to test for, and report correctly, all of the errors and warnings listed in Appendix E. A select group of test cases can be found in Appendix F. The complete test suite is available as part of the implementation[2].

With the compilation process completed and validated, the successful integration of the OCL with the C# language is complete. The next chapter will examine some of the specific issues that were discovered during the integration process.

Chapter 7

Issues

During the design and implementation of the compiler, several issues were discovered. The issues have been broken down into two categories: OCL related and implementation related. The OCL related category includes issues that are a direct result of the OCL specification. These issues will occur in all work where the OCL is used. The implementation related category includes issues that are specific to the integration of the OCL with C#. These issues may not necessarily exist when integrating the OCL with other implementation languages.

7.1 OCL Issues

The OCL has several issues that either create problems with implementing because of their ambiguity, or restrict the way in which the OCL can be used. This section will outline a few of the major issues with respect to the OCL specification. For a more complete list of outstanding issues in the OCL 2.0 specification, please see [30]¹.

¹The reader should note that the final OCL 2.0 specification has been accepted and is in its finalization phase. Thus, small changes may still occur.

7.1.1 Character Sets

The OCL specification refers to the ASCII character set. The ASCII character set does not support international character sets. The character set used in the OCL should be Unicode or a recognized international character set. Using an international character set will extend the usage of the OCL. Without an international character set, the OCL cannot be applied to models that contain text in languages that cannot be expressed using the ASCII character set.

In terms of implementation, the character set issue also comes into play when the OCL is being integrated with a language that uses a non-ASCII character set. C# is such a language, as it uses the Unicode character set.

In order to integrate the OCL into C#, our implementation of the OCL makes use of the Unicode character set. As our implementation of the OCL was done using C#, the use of the Unicode character set in the OCL required only a single modification to the OCL itself. The modification was to the OCL “String” type. In order to retain compatibility with C#, Unicode characters were integrated into the implementation of the OCL “String” type. This allows Unicode characters to be used in OCL string literals.

As each of the OCL expression was represented by a series of C# string literals, the entire OCL expression was able to contain Unicode characters. The use of Unicode characters in an OCL expression allows the expression to refer to types and members, which are defined using Unicode characters.

7.1.2 Null Values

The OCL specification does not allow for null, or empty, values. Most object-oriented languages allow for null values. In the case of the OCL, the notation of an undefined object is used. However, an undefined object is not always the same as a null object. An object could be defined to be a reference pointing to null.

C# does explicitly define the use of null values, and contains a specialized *null* keyword for assigning a null value. In order to facilitate the use of null values in the OCL, and to allow for maximum flexibility,

the compiler will implicitly convert between the OCL undefined object, and the C# *null* keyword as needed.

The solution to the null value issue present in the OCL specification is not a complete solution. While the compiler will convert between the two types as needed, it is possible that some of the strong type checking implied in the OCL specification will be lost. Our solution provides for maximum flexibility, but slightly reduces type safety. As C# is a strongly typed language, the use of a null value in the OCL will not result in a compile or runtime error. However, a constraint may fail in a case where it was not intended. The developer should take caution when using null values.

7.1.3 Missing Collection Operators

In the OCL type library, the notion of various collection types is defined. Only the non-abstract collection types define equality operators. The abstract collection type should also define such equality operators. As collections can be referenced via the abstract collection type, the equality operators should be available to compare abstract collections. The equality operators consist of an equality operator (“=”), and an inequality operator (“<>”).

Our implementation of the OCL type library includes the definition of both equality operators on the abstract “Collection” type. Two abstract collections will be said equal if they contain the same number of elements, and that each element present in collection one, is present in collection two. The C# implementation of the two equality operators is shown in Figure 7.1.

7.1.4 Keywords

The OCL specification defines several OCL keywords that cannot be used anywhere in an OCL expression. If the model on which the OCL is operating contains an attribute or operation that has the same name as an OCL keyword, the attribute or operation cannot be used. Table 7.1² lists the OCL keywords used in the implementation.

²Please note that * indicates a C# specific extension keyword.

```

#region == operator
/// <summary>
/// Returns true if each set contains the same elements
/// </summary>
/// <param name="s1">The first set to compare</param>
/// <param name="s2">The second set to compare</param>
/// <returns>True if each set contains the same elements</returns>
public static OclBoolean operator==(Collection s1, Collection s2)
{
    if((s1 as Object)== null || (s2 as Object) == null)
        return OclAny.BooleanUndefined;
    if(s1.oclIsUndefined() || s2.oclIsUndefined())
        return OclAny.BooleanUndefined;
    if(s1.size() == s2.size())
    {
        foreach(object o in s1._col)
            if(!s2._col.Contains(o))
                return new OclBoolean(false);
        return new OclBoolean(true);
    }
    return new OclBoolean(false);
}
#endregion
#region != operator
/// <summary>
/// Returns false if each set contains the same elements
/// </summary>
/// <param name="s1">The first set to compare</param>
/// <param name="s2">The second set to compare</param>
/// <returns>False if each set contains the same elements</returns>
public static OclBoolean operator!=(Collection s1, Collection s2)
{
    if(s1 == null || s2 == null)
        return OclAny.BooleanUndefined;
    if(s1.oclIsUndefined() || s2.oclIsUndefined())
        return OclAny.BooleanUndefined;
    if(s1.size() == s2.size())
    {
        foreach(object o in s1._col)
            if(!s2._col.Contains(o))
                return new OclBoolean(true);
        return new OclBoolean(false);
    }
    return new OclBoolean(true);
}
#endregion

```

Figure 7.1: Implementation of the equality operators on the *Collection* type

The user should not be restricted in naming attributes or operations because the constraint language does not permit the use of a given name. We believe that the usage restriction on the OCL keywords should be lifted to allow for maximum flexibility. However, in order to reduce the complexity of the grammar, our implementation does not lift this restriction. C# elements that are referenced via OCL cannot have the same name as any of the keywords in Table 7.1. The restriction only applies to C# elements if they

and	attr	Bag	body	Collection
context	def	derive	div	else
endif	endpackage	false	get*	if
implies	in	indexer*	init	inv
iterate	let	mod	not	oper
or	OrderedSet	package	pre	post
Sequence	Set	set*	then	true
Tuple	TupleType	xor		

Table 7.1: *The OCL keyword list*

are referenced in an OCL expression. If a C# element has a name of an OCL keyword, the element can still be used as long as it is not referenced in an OCL expression. Future work on the implementation should remove this restriction.

7.1.5 Ambiguous Grammar

As discussed in Chapter 3, some of the OCL grammatical elements found in the OCL specification cannot be constructed without using semantic information. For example, the type of an OCL expression determines if the expression denotes an attribute, an association end, or an operation.

The ambiguous grammar issue arises during the construction of an OCL implementation. Standard implementation of a language involves using a parser to produce an abstract syntax tree. The abstract syntax tree is then fed into a semantic analyzer, which in turn augments the abstract syntax tree by computing, for each node in the tree, the values of any attached attributes. The semantic analyzer also checks if there are any semantic errors.

In the OCL, the abstract syntax tree cannot be created directly from the grammar. Not being able to create the abstract syntax tree from the grammar creates a serious limitation when designing any implementation. Having such a hurdle in creating implementations for the OCL specification will reduce the number of OCL implementations. Other constraint languages may be selected based solely upon ease of implementation. Chapter 3 and Section 5.3.2 outline how the proposed implementation addresses the ambiguous grammar issue.

7.2 Implementation Issues

In addition to the issues presented in the previous section, there are issues that have risen directly from the integration of the OCL with C#. The issues presented in this section may not exist in other implementation languages.

7.2.1 Ambiguous Grammar

One of the major hurdles in the integration of the OCL with C# resulted from the grammar ambiguities stated in Chapter 3. Parsing of the OCL grammar cannot be accomplished without having access to a structural model of the system. The grammatical issue makes it impossible to build a complete abstract syntax tree from a standard parser generator. To avoid having to write a specialized parser, without the aid of a generator, each of the ambiguous grammatical elements were mapped into the single construct. At the beginning of the OCL semantic pass, the true meaning of the construct can be determined by examining the structural model. The structural model is created once the C# compiler has completed building the type hierarchy.

Each construct that can represent more than one element is resolved against the structural model to determine the corresponding element that should be used in the parse tree. The temporary parse tree node is then replaced with the correct element. Once all of the temporary constructs have been replaced with proper constructs, the semantic pass is executed.

The ambiguous grammar added several days of work with respect to the implementation. If the OCL specification contained a complete non-ambiguous grammar as in Appendix D, the integration would be easier in terms of both time and complexity.

7.2.2 allInstances

The OCL defines an “allInstances” operation on each type. The “allInstances” operation is defined to return a collection of all of the instances of the given type. As C# maintains an automatic garbage

collector, it is difficult to determine when an instance of a class has actually gone out of scope. In addition, there is no mechanism under C# to get the active instance list for a given type.

In order to implement the “allInstances” operation, specialized code would have to be added to each type, in order to keep track of the creation of new instances, the active list of instances, and the destruction of instances. This specialized code would introduce a significant decrease in performance.

In addition to the technical reasons, there is a semantic reason why the use of “allInstances” is discouraged. Consider the invariant in Figure 7.2. The invariant states that a person may not have more than two parents. The same expression may be expressed in Figure 7.3. As shown in the two figures, the use of the “allInstances” operation can make a simple invariant complex. The “allInstances” operation can sometimes hide the actual invariant, as is the case in Figure 7.2.

```
context Person
inv: Person.allInstances->forall(p | p.parents->size <= 2)
```

Figure 7.2: Using *allInstances* to state that a person may not have more than two parents

```
context Person
inv: parents->size <= 2
```

Figure 7.3: An OCL expression that states a person may not have more than two parents

Due to the technical and semantic issues, the “allInstances” operation is not supported in our implementation. If an occurrence of the “allInstances” operation is found, the compiler will issue a compile-time error.

7.2.3 Primitive Types

As stated in Chapter 5, the OCL numeric primitive types, “Integer”, and “Real”, have been mapped to the C# types, “int” and “double”, respectively. If the developer specifies a numeric value that is outside the range of the C# types, a compile-time warning is generated. The warning indicates that the value has been truncated so that it fits into the smaller type. In the case where two numeric types are used in a context where the result of an operation will result in a value that is outside of the types range, the value of the type will be undefined.

Ideally, the implementation should contain flexible “Integer” and “Real” OCL types, allowing values of any size to be represented. As our implementation is academic in nature, we postponed this change for future work.

In addition to the numeric types, the OCL “String” type is defined to use only ASCII characters. As stated before, Unicode characters were integrated into the implementation of the OCL “String” type.

It is expected that any integration between the OCL and another language will have issues mapping the language’s primitive data types to the four OCL types. One major drawback is that the OCL does not define a character type. In order to use the C# character type in the context of an OCL expression, the implementation maps the C# character type to the OCL “Integer” type.

7.2.4 Properties and Indexers

As illustrated in Chapters 2 and 3, C# contains some unique constructs that do not exist in other programming languages. In the context of integration with the OCL, the constructs that pose the most interest are properties and indexers. Properties and indexers define get and set accessors. An accessor is analogous to an operation. However accessors are not referenced as operations; in that they do not contain parenthesis at the end of the property or indexer name. To this end, the OCL specification does not allow preconditions or postconditions to be specified on accessors.

As properties, indexers and their accessors play an integral part in a C# application. Additional C# specific keywords have been added to the implementation of the OCL. The additional keywords allow for the specification of preconditions and postconditions on accessors that are defined within properties or accessors.

7.3 Summary

During the integration of any two independent technologies into a single technology, various issues always arise. The integration of the OCL with C# was no different. The use of the second version of the

OCL also added some additional issues. The OCL related issues are a result of the youth of the OCL. At the time of writing, there exist few OCL 2.0 implementations. None of these implementations makes use of a .NET compatible language. As the OCL specification matures, the OCL specific issues should be addressed, and the implementer should only have to worry about issues relating to the bridging of different technologies.

Chapter 8

Conclusion and Future Work

This chapter will present the current state of the implementation and outline the contributions made by this work. Following the contributions made by this work, an outline for future work will be discussed. The chapter will finish with some concluding statements.

8.1 Contributions

The research presented in this thesis, and the corresponding implementation, provide significant contributions in three areas of software engineering. The first area is in the context of the OCL itself. As to the best of our knowledge only two other OCL 2.0 compliant implementations exist, the implementation presented here adds to the validity of the OCL, providing a concrete implementation of the specification[4, 19]. Such an implementation can be viewed as a form of verification on the OCL specification[32].

The second area of contribution is in the context of MDA. As illustrated in Chapter 2, for MDA to be used effectively, the model transformations need to be automated. By adding OCL support at the code level, OCL expressions that exist in higher-level models can simply be propagated to the code level. The propagation step consists of a simple copy and paste operation, and reduces the complexity of MDA model transformations. Research in the area of propagating OCL expressions from UML models

to our implementation is ongoing. Toby McClean at Carleton University has devised a tool to provide such a bridge[24].

The third, and final area of contribution, is to the notion of design by contract. Allowing programmers to express constraints on various program elements using a formal language, adds the benefits of design by contract. The implementation allows constraints to be expressed separately from the implementation. Constraint support in a mainstream programming language (C#) facilitates the use of the design by contract methodology.

8.2 Future Work

Future work for this thesis falls into two categories. The first category is future work related to the implementation. The second category looks at future work in the context of the OCL.

8.2.1 Implementation

In terms of the implementation of the OCL specification[32], the implementation is feature complete with the exception of two elements. The first element is the implementation of the “allInstances” operation. For the reasons outlined in Chapter 7, the “allInstances” operation was not implemented. The second element that is not present in the implementation is the “OclMessage” type and its supporting operations. These operations include the “isSent” operator (^) and the “message” operator (^).

The main driver for not implementing the “OclMessage” type was that the implementation of some of the related operations is non-trivial in C#. In C#, it is difficult to determine the instance from which an operation is called. This issue would have to be resolved in order to implement the “OclMessage” type. But such a solution, which would be very language specific, is not easily obtained and thus lies beyond the scope of this work.

Furthermore, beyond the OCL considerations, the implementation is not guaranteed to be thread safe. Consider the case when a class invariant is invalidated during the execution of a method. If before the

invariant can be re-validated a context switch occurs, it is possible that a different method will test the same invariant, raising an exception.

Modifying of our implementation to support multi-threading environments is a non-trivial exercise. The compiler would have to determine when such a condition would occur, then introduce supporting critical sections to ensure that the constraints are checked in the proper order. A second solution could be to raise a warning and place the resolution of the multi-threading issue in the developer's hands.

Our extensive test suite was developed solely to validate our OCL compiler against the OCL specification[32]. Future work could adopt our test suite into a standard, language independent suite. The standard test suite could be used to test different OCL implementations and to verify their compliance with the OCL specification.

The final element of future work with respect to the implementation is optimization. As the compiler was designed as a proof of concept, several compiler optimization techniques could be applied to our compiler.

With various directions for future work at the implementation level outlined, we will now examine some directions for future work in the context of the OCL.

8.2.2 OCL

The OCL is a formal OMG specification. While several issues presented in Chapter 7 could be addressed directly at the specification level, we do not recommend them for future work as the OCL specification has been accepted.

In terms of the OCL, future work lies in the adopting of the OCL in different contexts. The research presented here only applies to the use of OCL with the C# language. Future integrations of the OCL with other technologies, will aid in the use of the OCL, as well as the raising of additional issues that relate directly to the OCL specification.

8.3 Conclusion

The use of the OCL in various UML models is an integral part of expressing and defining software systems. We strongly believe that the use of models for the expression and design of complex software systems is the way of the future, and once transformations between the various levels of models can be fully automated, models will play a larger picture in software development. These transformations form the basis of MDA, and by having a concrete code level method of expressing the OCL, the complexity of the actual transformation can be reduced. The implementation proposed in this dissertation can be seen as a step towards such automation.

Looking at this research from the code level has a different effect. Developers today tend to work only at the code level, and rarely at the documentation or modeling level. Adding a new formal constraint language to a process where code is the only language of expression, may not, in practice have the desired effect. For example, developers who are familiar with using code as their only means of expression, will use the implementation language to perform the function of contracts. These functions include verification of parameters, system state, and return values. In order to use OCL constraints, the developer will have to remove such validation from the implementation language, and replace it using the OCL.

In summary, the use of the OCL, either by itself or integrated with other technologies, is, and will be, a fundamental aspect of the software development process. However, if the use of the OCL is restricted to only the code level, it may be difficult for experienced developers to see the advantage of its use.

Bibliography

- [1] D. H. Akehurst and B. Bordbar. On Querying UML data models with OCL. Technical report, University of Kent, July 11 2001.
- [2] D. Arnold. C#/OCL Compiler, <http://www.ewesimplex.ca/csocl>.
- [3] K. Arnout and R. Simon. The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel. In *Technology of Object-Oriented Languages and Systems*, 2001.
- [4] Babes - Bolyai University. Object constraint language environment, <http://lci.cs.ubbcluj.ro/ocle/overview.html>.
- [5] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1st edition, 2000.
- [6] DotGNU Project. <http://www.dotgnu.org>.
- [7] H. Eriksson and M. Pemker. *Business Modeling with UML, Business Patterns at Work*. John Wiley & Sons, 2000.
- [8] R. Findler. *Behavioral Software Contracts*. Rice University, Houston, Texas, 2002.
- [9] D. Frankel. *Model Driven Architecture : Applying MDA to Enterprise Computing*. Wiley, New York, 2003.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

- [11] Hewlett-Packard, Intel, and Microsoft. C# Language Specification. Technical report, ECMA, December 2002.
- [12] Hewlett-Packard, Intel, and Microsoft. Common Language Infrastructure (CLI) Partition I: Concepts and Architecture . Technical report, ECMA, January 2002.
- [13] Hewlett-Packard, Intel, and Microsoft. Common Language Infrastructure (CLI): Partition II: Metadata Definition and Semantics. Technical report, ECMA, January 2002.
- [14] Hewlett-Packard, Intel, and Microsoft. Common Language Infrastructure (CLI): Partition III: CIL Instruction Set. Technical report, ECMA, January 2002.
- [15] Hewlett-Packard, Intel, and Microsoft. Common Language Infrastructure (CLI): Partition IV: Profiles and Libraries. Technical report, ECMA, January 2002.
- [16] Hewlett-Packard, Intel, and Microsoft. Common Language Infrastructure (CLI): Partition V: Annexes. Technical report, ECMA, January 2002.
- [17] D. Jackson. A Comparison of Object Modelling Notations: Alloy, UML, Z. Technical report, MIT, August 1999.
- [18] D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, MIT, February 2002.
- [19] Klasse Objecten. The octopus tool, <http://www.klasse.nl/ocl/octopus-intro.html>.
- [20] R. Kramer. iContract - the Java Design by Contract Tool. Technical report, Reliable Systems, 1998.
- [21] R. Leino. Private communication. Microsoft Research, April 2004.
- [22] R. Leino. Toward enforceable contracts for .NET. In *CASSIS*, March 2004.
- [23] S. Lidin. *Inside Microsoft .NET IL assembler*. Microsoft Press, Redmond, Wash., 2002.
- [24] T. McClean. Private communication. Carleton University, May 2004.

- [25] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, N.J., 2nd edition, 1997.
- [26] Microsoft. *Microsoft Developer Network Online Reference*, 2003.
- [27] Microsoft Research. Software productivity tools, <http://research.microsoft.com/spt>.
- [28] MIT Laboratory for Computer Science. Aloca: Alloy's constraint analyzer, <http://sdg.lcs.mit.edu/alloy>.
- [29] R. Mitchell and J. McKim. *Design by contract, by example*. Addison Wesley, Boston, MA, 2002.
- [30] Object Management Group: OCL 2.0 Finalization Task Force. Issues for OCL 2.0 Finalization Task Force, <http://www.omg.org/issues/ocl2-fff.html>.
- [31] OMG. UML 2.0 OCL RFP. Technical report, OMG, September 9 2000. OMG Document: ad/2000-09-03.
- [32] OMG. Response to the UML 2.0 OCL RfP. Technical report, OMG, January 6th 2003. OMG document ad2003-01-16.
- [33] OMG. UML 2.0 Infrastructure Specification. Technical report, OMG, September 15th 2003.
- [34] OMG. UML 2.0 Superstructure Specification. Technical report, OMG, August 2nd 2003.
- [35] OMG. MOF 2.0 Core Specification. Technical report, OMG, March 10th 2004.
- [36] OMG Web Site. <http://www.omg.org>.
- [37] Parasoft. Jcontract, <http://www.parasoft.com/jsp/products/home.jsp>.
- [38] J. Prosise. *Programming Microsoft .NET (core reference)*. Microsoft Press : Distributed in Canada by Penguin Books Canada Limited, Redmond, Wash., 2002.
- [39] SSCLI Web Site. <http://www.sscli.net>.
- [40] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly & Associates, Sebastopol, 2003.

-
- [41] The Eclipse Foundation. The eclipse framework, <http://www.eclipse.org>.
 - [42] The Eiffel Programming Language. <http://www.eiffel.com>.
 - [43] J. Trupin. Sharp New Language: C# Offers the Power of C++ and Simplicity of Visual Basic. *MSDN Magazine*, 2002(Sept.), 2002.
 - [44] M. Vaziri and D. Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. Technical report, MIT, December 1999.
 - [45] J. Warmer and A. Kleppe. *The Object Constraint Language : Getting Your Models Ready for MDA*. Addison-Wesley, Boston, MA, 2003.
 - [46] M. Williams. *Microsoft Visual C# .NET*. Microsoft Press, Redmond, Wash., 2002.
 - [47] Ximian - Mono Project. <http://developer.ximian.com/projects/mono/>.

Appendix A

List of Acronyms

Acronyms	Definition
API	Application Programming Interface
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COFF	Common Object File Format
COM	Component Object Model
FCL	Framework Class Library
IL	Intermediate Language
JIT	Just In Time
MDA	Model Driven Architecture
MFC	Microsoft Foundation Classes
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PE	Portable Executable
PIM	Platform Independent Model
PSM	Platform Specific Model
SQL	Structured Query Language
SSCLI	Shared Source Common Language Infrastructure
STL	Standard Template Library
XMI	XML Metadata Interchange
XML	Extensible Markup Language
UML	Unified Modeling Language

Table A.2: *List of Acronyms*

Appendix B

Colour Plates

Figure B.1: *OCL constraint failure dialog box*

Figure B.3: C#/OCL compiler structure

Appendix C

C# Grammar to support the OCL

This appendix contains the lexical and syntactic grammars used for the C# parser. The modifications required to support the OCL blocks have been added to the grammar. The original grammar is based on the grammar found in the ECMA C# specification[11].

C.1 Lexical grammar

```
input:
    input-sectionopt
input-section:
    input-section-part
    input-section input-section-part
input-section-part:
    input-elementsopt new-line
    pp-directive
input-elements:
    input-element
    input-elements input-element
input-element:
    whitespace
    comment
    token
```

C.1.1 Line terminators

```
new-line:
    Carriage return character (U+000D)
```

Line feed character (U+000A)
Carriage return character (U+000D) followed by line
feed character (U+000A)
Line separator character (U+2028)
Paragraph separator character (U+2029)

C.1.2 White space

whitespace:
Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)
Form feed character (U+000C)

C.1.3 Comments

comment:
 single-line-comment
 delimited-comment
single-line-comment:
 // *input-characters_{opt}*
input-characters:
 input-character
 input-characters input-character
input-character:
 Any Unicode character except a *new-line-character*
new-line-character:
 Carriage return character (U+000D)
 Line feed character (U+000A)
 Line separator character (U+2028)
 Paragraph separator character (U+2029)
delimited-comment:
 /* *delimited-comment-characters_{opt}* */
delimited-comment-characters:
 delimited-comment-character
 delimited-comment-characters delimited-comment-character
delimited-comment-character:
 not-asterisk
 * *not-slash*
not-asterisk:
 Any Unicode character except *
not-slash:
 Any Unicode character except /

C.1.4 Tokens

token:
 identifier

```

keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

```

C.1.5 Unicode character escape sequences

```

unicode-escape-sequence:
    \u hex-digit hex-digit hex-digit hex-digit
    \U hex-digit hex-digit hex-digit hex-digit
        hex-digit hex-digit hex-digit hex-digit

```

C.1.6 Identifiers

```

identifier:
    available-identifier
    @ identifier-or-keyword
available-identifier:
    An identifier-or-keyword that is not a keyword
identifier-or-keyword:
    identifier-start-character identifier-part-charactersopt
identifier-start-character:
    letter-character
    _ (the underscore character U+005F)
identifier-part-characters:
    identifier-part-character
    identifier-part-characters identifier-part-character
identifier-part-character:
    letter-character
    decimal-digit-character
    connecting-character
    combining-character
    formatting-character
letter-character:
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
    A unicode-escape-sequence representing a character of classes
        Lu, Ll, Lt, Lm, Lo, or Nl
combining-character:
    A Unicode character of classes Mn or Mc
    A unicode-escape-sequence representing a character of classes
        Mn or Mc
decimal-digit-character:
    A Unicode character of the class Nd
    A unicode-escape-sequence representing a character of the
        class Nd
connecting-character:
    A Unicode character of the class Pc
    A unicode-escape-sequence representing a character of the
        class Pc
formatting-character:

```

A Unicode character of the class Cf
 A unicode-escape-sequence representing a character of the
 class Cf

C.1.7 Keywords

keyword: one of

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	init	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while
OCL						

C.1.8 Literals

literal:
 boolean-literal
 integer-literal
 real-literal
 character-literal
 string-literal
 null-literal
boolean-literal:
 true
 false
integer-literal:
 decimal-integer-literal
 hexadecimal-integer-literal
decimal-integer-literal:
 decimal-digits integer-type-suffix_{opt}
decimal-digits:
 decimal-digit
 decimal-digits decimal-digit
decimal-digit: one of
 0 1 2 3 4 5 6 7 8 9
integer-type-suffix: one of
 U u L l UL Ul uL ul LU Lu lU lu
hexadecimal-integer-literal:
 0x hex-digits integer-type-suffix_{opt}
 0X hex-digits integer-type-suffix_{opt}
hex-digits:
 hex-digit

```

    hex-digits hex-digit
hex-digit: one of
    0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
real-literal:
    decimal-digits . decimal-digits exponent-partopt real-type-suffixopt
    . decimal-digits exponent-partopt real-type-suffixopt
    decimal-digits exponent-part real-type-suffixopt
    decimal-digits real-type-suffix
exponent-part:
    e signopt decimal-digits
    E signopt decimal-digits
sign: one of
    + -
real-type-suffix: one of
    F f D d M m
character-literal:
    ' character '
character:
    single-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence
single-character:
    Any character except ' (U+0027), \ (U+005C), and new-line-character
simple-escape-sequence: one of
    \' \" \\ \0 \a \b \f \n \r \t \v
hexadecimal-escape-sequence:
    \x hex-digit hex-digitopt hex-digitopt hex-digitopt
string-literal:
    regular-string-literal
    verbatim-string-literal
regular-string-literal:
    " regular-string-literal-charactersopt "
regular-string-literal-characters:
    regular-string-literal-character
    regular-string-literal-characters regular-string-literal-character
regular-string-literal-character:
    single-regular-string-literal-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence
single-regular-string-literal-character:
    Any character except " (U+0022), \ (U+005C), and new-line-character
verbatim-string-literal:
    @" verbatim-string-literal-charactersopt "
verbatim-string-literal-characters:
    verbatim-string-literal-character
    verbatim-string-literal-characters verbatim-string-literal-character
verbatim-string-literal-character:
    single-verbatim-string-literal-character
    quote-escape-sequence
single-verbatim-string-literal-character:
    any character except "
quote-escape-sequence:
    ""
null-literal:
    null

```

C.1.9 Operators and punctuators

```

operator-or-punctuator: one of
{ } [ ] ( ) . , : ;
+ - * / % & | ^ ! ~
= < > ? ++ -- && || << >>
== != <= >= += -= *= /= %= &=
|= ^= <<= >>= ->

```

C.1.10 Pre-processing directives

```

pp-directive:
  pp-declaration
  pp-conditional
  pp-line
  pp-diagnostic
  pp-region
pp-new-line:
  whitespaceopt single-line-commentopt new-line
conditional-symbol:
  Any identifier-or-keyword except true or false
pp-expression:
  whitespaceopt pp-or-expression whitespaceopt
pp-or-expression:
  pp-and-expression
  pp-or-expression whitespaceopt || whitespaceopt pp-and-expression
pp-and-expression:
  pp-equality-expression
  pp-and-expression whitespaceopt && whitespaceopt
  pp-equality-expression
pp-equality-expression:
  pp-unary-expression
  pp-equality-expression whitespaceopt == whitespaceopt
  pp-unary-expression
  pp-equality-expression whitespaceopt != whitespaceopt
  pp-unary-expression
pp-unary-expression:
  pp-primary-expression
  ! whitespaceopt pp-unary-expression
pp-primary-expression:
  true
  false
  conditional-symbol
  ( whitespaceopt pp-expression whitespaceopt )
pp-declaration:
  whitespaceopt # whitespaceopt define whitespace
  conditional-symbol pp-new-line
  whitespaceopt # whitespaceopt undef whitespace
  conditional-symbol pp-new-line
pp-conditional:
  pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif
pp-if-section:
  whitespaceopt # whitespaceopt if whitespace pp-expression
  pp-new-line conditional-sectionopt
pp-elif-sections:

```



```

    pp-elif-section
    pp-elif-sections pp-elif-section
pp-elif-section:
    whitespaceopt # whitespaceopt elif whitespace
    pp-expression pp-new-line conditional-sectionopt
pp-else-section:
    whitespaceopt # whitespaceopt else pp-new-line
    conditional-sectionopt
pp-endif:
    whitespaceopt # whitespaceopt endif pp-new-line
conditional-section:
    input-section
    skipped-section
skipped-section:
    skipped-section-part
    skipped-section skipped-section-part
skipped-section-part:
    skipped-charactersopt new-line
    pp-directive
skipped-characters:
    whitespaceopt not-number-sign input-charactersopt
not-number-sign:
    Any input-character except #
pp-line:
    whitespaceopt # whitespaceopt line whitespace line-indicator
    pp-new-line
line-indicator:
    decimal-digits whitespace file-name
    decimal-digits
    default
file-name:
    " file-name-characters "
file-name-characters:
    file-name-character
    file-name-characters file-name-character
file-name-character:
    Any input-character except "
pp-diagnostic:
    whitespaceopt # whitespaceopt error pp-message
    whitespaceopt # whitespaceopt warning pp-message
pp-message:
    new-line
whitespace input-charactersopt new-line
pp-region:
    pp-start-region conditional-sectionopt pp-end-region
pp-start-region:
    whitespaceopt # whitespaceopt region pp-message
pp-end-region:
    whitespaceopt # whitespaceopt endregion pp-message

```

C.2 Syntactic grammar

C.2.1 Basic concepts

```
namespace-name:  
    namespace-or-type-name  
type-name:  
    namespace-or-type-name  
namespace-or-type-name:  
    identifier  
    namespace-or-type-name . identifier
```

C.2.2 Types

```
type:  
    value-type  
    reference-type  
value-type:  
    struct-type  
    enum-type  
struct-type:  
    type-name  
    simple-type  
simple-type:  
    numeric-type  
    bool  
numeric-type:  
    integral-type  
    floating-point-type  
    decimal  
integral-type:  
    sbyte  
    byte  
    short  
    ushort  
    int  
    uint  
    long  
    ulong  
    char  
floating-point-type:  
    float  
    double  
enum-type:  
    type-name  
reference-type:  
    class-type  
    interface-type  
    array-type  
    delegate-type  
class-type:  
    type-name
```

```
object
string
interface-type:
    type-name
array-type:
    non-array-type rank-specifiers
non-array-type:
    type
rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier
rank-specifier:
    [ dim-separatorsopt ]
dim-separators:
    '
    dim-separators ,
delegate-type:
    type-name
```

C.2.3 Variables

```
variable-reference:
    expression
```

C.2.4 Expressions

```
argument-list:
    argument
    argument-list , argument
argument:
    expression
    ref variable-reference
    out variable-reference
primary-expression:
    primary-no-array-creation-expression
    array-creation-expression
    primary-no-array-creation-expression:
        literal
        simple-name
        parenthesized-expression
        member-access
        invocation-expression
        element-access
        this-access
        base-access
        post-increment-expression
        post-decrement-expression
        object-creation-expression
        delegate-creation-expression
        typeof-expression
        sizeof-expression
        checked-expression
```

```

unchecked-expression
simple-name:
    identifier
parenthesized-expression:
    ( expression )
member-access:
    primary-expression . identifier
    predefined-type . identifier
predefined-type: one of
    bool byte char decimal double float int long
    object sbyte short string uint ulong ushort
invocation-expression:
    primary-expression ( argument-listopt )
element-access:
    primary-no-array-creation-expression [ expression-list ]
expression-list:
    expression
    expression-list , expression
this-access:
    this
base-access:
    base . identifier
    base [ expression-list ]
post-increment-expression:
    primary-expression ++
post-decrement-expression:
    primary-expression --
object-creation-expression:
    new type ( argument-listopt )
array-creation-expression:
    new non-array-type [ expression-list ] rank-specifiersopt
    array-initializeropt
    new array-type array-initializer
delegate-creation-expression:
    new delegate-type ( expression )
typeof-expression:
    typeof ( type )
    typeof ( void )
checked-expression:
    checked ( expression )
unchecked-expression:
    unchecked ( expression )
unary-expression:
    primary-expression
    + unary-expression
    - unary-expression
    ! unary-expression
    ~ unary-expression
    * unary-expression
    pre-increment-expression
    pre-decrement-expression
    cast-expression
pre-increment-expression:
    ++ unary-expression
pre-decrement-expression:
    -- unary-expression
cast-expression:
    ( type ) unary-expression
multiplicative-expression:

```

```

unary-expression
multiplicative-expression * unary-expression
multiplicative-expression / unary-expression
multiplicative-expression % unary-expression
additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression
shift-expression:
additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression
relational-expression:
shift-expression
relational-expression < shift-expression
relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression
relational-expression is type
relational-expression as type
equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression
and-expression:
equality-expression
and-expression & equality-expression
exclusive-or-expression:
and-expression
exclusive-or-expression ^ and-expression
inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression | exclusive-or-expression
conditional-and-expression:
inclusive-or-expression
conditional-and-expression && inclusive-or-expression
conditional-or-expression:
conditional-and-expression
conditional-or-expression || conditional-and-expression
conditional-expression:
conditional-or-expression
conditional-or-expression ? expression : expression
assignment:
unary-expression assignment-operator expression
assignment-operator: one of
= += -= *= /= %= &= |= ^= <=> >=>
expression:
conditional-expression
assignment
constant-expression:
expression
boolean-expression:
expression

```

C.2.5 Statements

```
statement:
    labeled-statement
    declaration-statement
    embedded-statement
embedded-statement:
    block
    empty-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
    try-statement
    checked-statement
    unchecked-statement
    lock-statement
    using-statement
block:
    { statement-listopt }
statement-list:
    statement
    statement-list statement
empty-statement:
    ;
labeled-statement:
    identifier : statement
declaration-statement:
    local-variable-declaration ;
    local-constant-declaration ;
local-variable-declaration:
    type local-variable-declarators
local-variable-declarators:
    local-variable-declarator
    local-variable-declarators , local-variable-declarator
local-variable-declarator:
    identifier
    identifier = local-variable-initializer
local-variable-initializer:
    expression
    array-initializer
local-constant-declaration:
    const type constant-declarators
constant-declarators:
    constant-declarator
    constant-declarators , constant-declarator
constant-declarator:
    identifier = constant-expression
expression-statement:
    statement-expression ;
statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression
```

```

selection-statement:
    if-statement
    switch-statement
if-statement:
    if ( boolean-expression ) embedded-statement
    if ( boolean-expression ) embedded-statement else embedded-statement
boolean-expression:
    expression
switch-statement:
    switch ( expression ) switch-block
switch-block:
    { switch-sectionsopt }
switch-sections:
    switch-section
    switch-sections switch-section
switch-section:
    switch-labels statement-list
switch-labels:
    switch-label
    switch-labels switch-label
switch-label:
    case constant-expression :
    default :
iteration-statement:
    while-statement
    do-statement
    for-statement
    foreach-statement
while-statement:
    while ( boolean-expression ) embedded-statement
do-statement:
    do embedded-statement while ( boolean-expression ) ;
for-statement:
    for ( for-initializeropt ; for-conditionopt ; for-iteratoropt )
        embedded-statement
for-initializer:
    local-variable-declaration
    statement-expression-list
for-condition:
    boolean-expression
for-iterator:
    statement-expression-list
statement-expression-list:
    statement-expression
    statement-expression-list , statement-expression
foreach-statement:
    foreach ( type identifier in expression ) embedded-statement
jump-statement:
    break-statement
    continue-statement
    goto-statement
    return-statement
    throw-statement
break-statement:
    break ;
continue-statement:
    continue ;
goto-statement:
    goto identifier ;

```

```

    goto case constant-expression ;
    goto default ;
return-statement:
    return expressionopt ;
throw-statement:
    throw expressionopt ;
try-statement:
    try block catch-clauses
    try block finally-clause
    try block catch-clauses finally-clause
catch-clauses:
    specific-catch-clauses general-catch-clauseopt
    specific-catch-clausesopt general-catch-clause
specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses specific-catch-clause
specific-catch-clause:
    catch ( class-type identifieropt ) block
general-catch-clause:
    catch block
finally-clause:
    finally block
checked-statement:
    checked block
unchecked-statement:
    unchecked block
lock-statement:
    lock ( expression ) embedded-statement
using-statement:
    using ( resource-acquisition ) embedded-statement
resource-acquisition:
    local-variable-declaration
    expression

```

C.2.6 Namespaces

```

compilation-unit:
    using-directivesopt ocl-expressionsopt global-attributesopt
    namespace-member-declarationsopt
namespace-declaration:
    namespace qualified-identifier namespace-body ;opt
qualified-identifier:
    identifier
    qualified-identifier . identifier
namespace-body:
    { using-directivesopt namespace-member-declarationsopt }
using-directives:
    using-directive
    using-directives using-directive
using-directive:
    using-alias-directive
    using-namespace-directive
using-alias-directive:
    using identifier = namespace-or-type-name ;
using-namespace-directive:

```



```

    using namespace-name ;
namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations namespace-member-declaration
namespace-member-declaration:
    namespace-declaration
    type-declaration
type-declaration:
    class-declaration
    struct-declaration
    interface-declaration
    enum-declaration
    delegate-declaration

```

C.2.7 Classes

```

class-declaration:
    ocl-expressionsopt attributesopt class-modifiersopt class
    identifier class-baseopt class-body ;opt
class-modifiers:
    class-modifier
    class-modifiers class-modifier
class-modifier:
    new
    public
    protected
    internal
    private
    abstract
    sealed
class-base:
    : class-type
    : interface-type-list
    : class-type , interface-type-list
interface-type-list:
    interface-type
    interface-type-list , interface-type
class-body:
    { class-member-declarationsopt }
class-member-declarations:
    class-member-declaration
    class-member-declarations class-member-declaration
class-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    destructor-declaration
    static-constructor-declaration
    type-declaration
constant-declaration:

```

```

    ocl-expressionsopt attributesopt constant-modifiersopt const type
    constant-declarators ;
constant-modifiers:
    constant-modifier
    constant-modifiers constant-modifier
constant-modifier:
    new
    public
    protected
    internal
    private
constant-declarators:
    constant-declarator
    constant-declarators , constant-declarator
constant-declarator:
    identifier = constant-expression
field-declaration:
    ocl-expressionsopt attributesopt field-modifiersopt type
    variable-declarators ;
field-modifiers:
    field-modifier
    field-modifiers field-modifier
field-modifier:
    new
    public
    protected
    internal
    private
    static
    readonly
    volatile
variable-declarators:
    variable-declarator
    variable-declarators , variable-declarator
variable-declarator:
    identifier
    identifier = variable-initializer
variable-initializer:
    expression
    array-initializer
method-declaration:
    method-header method-body
method-header:
    ocl-expressionsopt attributesopt method-modifiersopt return-type
    member-name ( formal-parameter-listopt )
method-modifiers:
    method-modifier
    method-modifiers method-modifier
method-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract

```

```

    extern
return-type:
    type
    void
member-name:
    identifier
    interface-type . identifier
method-body:
    block
    ;
formal-parameter-list:
    fixed-parameters
    fixed-parameters , parameter-array
    parameter-array
fixed-parameters:
    fixed-parameter
    fixed-parameters , fixed-parameter
fixed-parameter:
    ocl-expressionsopt attributesopt parameter-modifieropt type
    identifier
parameter-modifier:
    ref
    out
parameter-array:
    ocl-expressionsopt attributesopt params array-type identifier
property-declaration:
    ocl-expressionsopt attributesopt property-modifiersopt type
    member-name { accessor-declarations }
property-modifiers:
    property-modifier
    property-modifiers property-modifier
property-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern
member-name:
    identifier
    interface-type . identifier
accessor-declarations:
    get-accessor-declaration set-accessor-declarationopt
    set-accessor-declaration get-accessor-declarationopt
get-accessor-declaration:
    ocl-expressionsopt attributesopt get accessor-body
set-accessor-declaration:
    ocl-expressionsopt attributesopt set accessor-body
accessor-body:
    block
    ;
event-declaration:
    ocl-expressionsopt attributesopt event-modifiersopt event type
    variable-declarators ;

```

```

    ocl-expressionsopt attributesopt event-modifiersopt event type
    member-name { event-accessor-declarations }
event-modifiers:
    event-modifier
event-modifiers event-modifier
event-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern
event-accessor-declarations:
    add-accessor-declaration remove-accessor-declaration
    remove-accessor-declaration add-accessor-declaration
add-accessor-declaration:
    ocl-expressionsopt attributesopt add block
remove-accessor-declaration:
    ocl-expressionsopt attributesopt remove block
indexer-declaration:
    ocl-expressionsopt attributesopt indexer-modifiersopt
    indexer-declarator { accessor-declarations }
indexer-modifiers:
    indexer-modifier
    indexer-modifiers indexer-modifier
indexer-modifier:
    new
    public
    protected
    internal
    private
    virtual
    sealed
    override
    abstract
    extern
indexer-declarator:
    type this [ formal-parameter-list ]
    type interface-type . this [ formal-parameter-list ]
operator-declaration:
    ocl-expressionsopt attributesopt operator-modifiers
    operator-declarator operator-body
operator-modifiers:
    operator-modifier
    operator-modifiers operator-modifier
operator-modifier:
    public
    static
    extern
operator-declarator:
    unary-operator-declarator
    binary-operator-declarator
    conversion-operator-declarator
unary-operator-declarator:

```

```

    type operator overloadable-unary-operator ( type identifier )
overloadable-unary-operator: one of
    + - ! ~ ++ -- true false
binary-operator-declarator:
    type operator overloadable-binary-operator ( type identifier ,
        type identifier )
overloadable-binary-operator: one of
    + - * / % & | ^ << >> == != > < >= <=
conversion-operator-declarator:
    implicit operator type ( type identifier )
    explicit operator type ( type identifier )
operator-body:
    block
    ;
constructor-declaration:
    ocl-expressionsopt attributesopt constructor-modifiersopt
        constructor-declarator constructor-body
constructor-modifiers:
    constructor-modifier
    constructor-modifiers constructor-modifier
constructor-modifier:
    public
    protected
    internal
    private
    extern
constructor-declarator:
    identifier ( formal-parameter-listopt ) constructor-initializeropt
constructor-initializer:
    : base ( argument-listopt )
    : this ( argument-listopt )
constructor-body:
    block
    ;
static-constructor-declaration:
    ocl-expressionsopt attributesopt static-constructor-modifiers
        identifier ( ) static-constructor-body
static-constructor-modifiers:
    externopt static
    static externopt
static-constructor-body:
    block
    ;
destructor-declaration:
    ocl-expressionsopt attributesopt externopt~ identifier ( )
        destructor-body
destructor-body:
    block
    ;

```

C.2.8 Structures

```

struct-declaration:
    ocl-expressionsopt attributesopt struct-modifiersopt struct
        identifier struct-interfacesopt struct-body ;opt

```

```

struct-modifiers:
    struct-modifier
    struct-modifiers struct-modifier
struct-modifier:
    new
    public
    protected
    internal
    private
struct-interfaces:
    : interface-type-list
struct-body:
{ struct-member-declarationsopt }
struct-member-declarations:
    struct-member-declaration
    struct-member-declarations struct-member-declaration
struct-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration

```

C.2.9 Arrays

```

array-type:
    non-array-type rank-specifiers
non-array-type:
    type
rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier
rank-specifier:
    [ dim-separatorsopt ]
dim-separators:
    ,
    dim-separators ,
array-initializer:
    { variable-initializer-listopt }
    { variable-initializer-list , }
variable-initializer-list:
    variable-initializer
    variable-initializer-list , variable-initializer
variable-initializer:
    expression
    array-initializer

```

C.2.10 Interfaces

```

interface-declaration:
    ocl-expressionsopt attributesopt interface-modifiersopt interface
        identifier interface-baseopt interface-body ;opt
interface-modifiers:
    interface-modifier
    interface-modifiers interface-modifier
interface-modifier:
    new
    public
    protected
    internal
    private
interface-base:
    : interface-type-list
interface-body:
    { interface-member-declarationsopt }
interface-member-declarations:
    interface-member-declaration
    interface-member-declarations interface-member-declaration
interface-member-declaration:
    interface-method-declaration
    interface-property-declaration
    interface-event-declaration
    interface-indexer-declaration
interface-method-declaration:
    ocl-expressionsopt attributesopt newopt return-type identifier
        ( formal-parameter-listopt ) ;
interface-property-declaration:
    ocl-expressionsopt attributesopt newopt type identifier
        { interface-accessors }
interface-accessors:
    ocl-expressionsopt attributesopt get ;
    ocl-expressionsopt attributesopt set ;
    ocl-expressionsopt attributesopt get ; ocl-expressionsopt
        attributesopt set ;
    ocl-expressionsopt attributesopt set ; ocl-expressionsopt
        attributesopt get ;
interface-event-declaration:
    ocl-expressionsopt attributesopt newopt event type identifier ;
interface-indexer-declaration:
    ocl-expressionsopt attributesopt newopt type this
        [ formal-parameter-list ] { interface-accessors }

```

C.2.11 Enums

```

enum-declaration:
    ocl-expressionsopt attributesopt enum-modifiersopt enum\
        identifier enum-baseopt enum-body ;opt
enum-base:
    : integral-type
enum-body:
    { enum-member-declarationsopt }
    { enum-member-declarations , }

```

```

enum-modifiers:
    enum-modifier
    enum-modifiers enum-modifier
enum-modifier:
    new
    public
    protected
    internal
    private
enum-member-declarations:
    enum-member-declaration
    enum-member-declarations , enum-member-declaration
enum-member-declaration:
    ocl-expressionsopt attributesopt identifier
    ocl-expressionsopt attributesopt identifier = constant-expression

```

C.2.12 Delegates

```

delegate-declaration:
    ocl-expressionsopt attributesopt delegate-modifiersopt delegate\
    return-type identifier ( formal-parameter-listopt ) ;
delegate-modifiers:
delegate-modifier
delegate-modifiers delegate-modifier
delegate-modifier:
    new
    public
    protected
    internal
    private

```

C.2.13 Attributes

```

global-attributes:
    global-attribute-sections
global-attribute-sections:
    global-attribute-section
    global-attribute-sections global-attribute-section
global-attribute-section:
    [ global-attribute-target-specifier attribute-list ]
    [ global-attribute-target-specifier attribute-list , ]
global-attribute-target-specifier:
    global-attribute-target :
global-attribute-target:
    assembly
    module
attributes:
    attribute-sections
attribute-sections:
    attribute-section
    attribute-sections attribute-section
attribute-section:

```



```

    [ attribute-target-specifieropt attribute-list ]
    [ attribute-target-specifieropt attribute-list , ]
attribute-target-specifier:
    attribute-target :
attribute-target:
    field
    event
    method
    param
    property
    return
    type
attribute-list:
    attribute
    attribute-list , attribute
attribute:
    attribute-name attribute-argumentsopt
attribute-name:
    type-name
attribute-arguments:
    ( positional-argument-listopt )
    ( positional-argument-list , named-argument-list )
    ( named-argument-list )
positional-argument-list:
    positional-argument
    positional-argument-list , positional-argument
positional-argument:
    attribute-argument-expression
named-argument-list:
    named-argument
    named-argument-list , named-argument
named-argument:
    identifier = attribute-argument-expression
attribute-argument-expression:
    expression

```

C.2.14 OCL

```

ocl_expression:
    ocl-sections
ocl-sections:
    ocl-section
    ocl-sections ocl-section
ocl-section:
    OCL [ ocl-bodyopt ]
ocl-body:
    string-literal
    ocl-body string-literal

```

C.3 Unsafe code

```

class-modifier:

```

```

...
unsafe
struct-modifier:
...
unsafe
interface-modifier:
...
unsafe
delegate-modifier:
...
unsafe
field-modifier:
...
unsafe
method-modifier:
...
unsafe
property-modifier:
...
unsafe
event-modifier:
...
unsafe
indexer-modifier:
...
unsafe
operator-modifier:
...
unsafe
constructor-modifier:
...
unsafe
destructor-declaration:
    ocl-expressionsopt attributesopt externopt unsafeopt ~
        identifier ( ) destructor-body
    ocl-expressionsopt attributesopt unsafeopt externopt ~
        identifier ( ' ) destructor-body
static-constructor-modifiers:
    externopt unsafeopt static
    unsafeopt externopt static
    externopt static unsafeopt
    unsafeopt static externopt
    static externopt unsafeopt
    static unsafeopt externopt
embedded-statement:
...
unsafe-statement
unsafe-statement:
    unsafe block
type:
    value-type
    reference-type
    pointer-type
pointer-type:
    unmanaged-type *
    void *
unmanaged-type:
    type
primary-no-array-creation-expression:

```

```

...
pointer-member-access
pointer-element-access
sizeof-expression
unary-expression:
...
pointer-indirection-expression
addressof-expression
pointer-indirection-expression:
    * unary-expression
pointer-member-access:
    primary-expression -> identifier
pointer-element-access:
    primary-no-array-creation-expression [ expression ]
addressof-expression:
    & unary-expression
sizeof-expression:
    sizeof ( unmanaged-type )
embedded-statement:
...
fixed-statement
fixed-statement:
    fixed ( pointer-type fixed-pointer-declarators ) embedded-statement
fixed-pointer-declarators:
    fixed-pointer-declarator
    fixed-pointer-declarators , fixed-pointer-declarator
fixed-pointer-declarator:
    identifier = fixed-pointer-initializer
fixed-pointer-initializer:
    & variable-reference
    expression
variable-initializer:
    expression
    array-initializer
    stackalloc-initializer
stackalloc-initializer:
    stackalloc unmanaged-type [ expression ]

```

Appendix D

OCL Grammar

This appendix contains the OCL grammar used as input to the JAY parser generator[47]. For more information see the implementation[2].

D.1 Grammar Header

```
%{  
// -----  
// OCL-parser.jay - The main OCL parser  
// -----  
// Project: C# and OCL Compiler  
// Module:  OCL Compiler  
// Author:  Dave Arnold  
// Version: 1.0  
// -----  
// -----  
// Imports  
// -----  
using System.Text;  
using System.IO;  
using System;  
using System.Collections;  
using DaveArnold.OCLCompiler.OCLElements;  
using DaveArnold.OCL2STL.PrimitiveTypes;  
// -----  
// -----  
// DaveArnold.OCLCompiler Namespace  
// -----  
namespace DaveArnold.OCLCompiler  
{  
    /// <summary>
```

```

    /// The OCL Parser
    /// </summary>
    public class OCLParser
    {
        // The current context declarator
        private ContextDecl current_context = null;

        // The current OCL expression
        private OclExpression current_expression = null;
    }

```

D.2 Tokens

```

%token EOF
%token NONE
%token ERROR
/*
 *These are the OCL keywords
 */
%token FIRST_KEYWORD
%token PACKAGE
%token ENDPACKAGE
%token CONTEXT
%token INV
%token DEF
%token PRE
%token POST
%token IMPLIES
%token AND
%token OR
%token NOT
%token TRUE
%token FALSE
%token XOR
%token IF
%token THEN
%token ELSE
%token ENDIF
%token LET
%token IN
%token ATTR
%token OPER
// Added Keywords For Easier Processing (NOT IN SPEC)
%token SET
%token BAG
%token SEQUENCE
%token ORDEREDSET
%token COLLECTION
%token TUPLETYPE
%token TUPLE
%token INIT
%token DERIVE
%token BODY
%token ITERATE
%token DIV

```

```

%token MOD
// Extension Keywords
%token EXT_GET
%token EXT_SET
%token EXT_INDEXER
%left LAST_KEYWORD

```

D.3 Punctuation

```

/* OCL single character operators/punctuation. */
%token OPEN_BRACE      "{"
%token CLOSE_BRACE     "}"
%token OPEN_BRACKET    "["
%token CLOSE_BRACKET   "]"
%token OPEN_PARENS     "("
%token CLOSE_PARENS    ")"
%token COMMA           ","
%token SEMICOLON       ";"
%token AT              "@"
%token QUESTION        "?"
%token PIPE            "|"
%token EQUALITY        "="
%token PLUS            "+"
%token MULTIPLY        "-"
%token DIVIDE          "/"
%token LESS            "<"
%token GREATOR         ">"
%token UP              "^"
%token DOT             "."
%token COLON           ":"
%token MINUS           "-"
/* OCL multi-character operators. */
%token DOT_DOT         ".."
%token SCOPE           "::"
%token ACCESSOR        "->"
%token UP_UP           "^@"
%token NOT_EQUALITY    "<>"
%token LESS_EQUAL      "<="
%token GREATOR_EQUAL   ">="
/* Literals */
%token LITERAL_INTEGER      "int literal"
%token LITERAL_REAL         "real literal"
%token LITERAL_STRING       "string literal"
%token IDENTIFIER

```

D.4 Precedence Rules

```

%nonassoc LOWPREC
%left COLON
%left UP UP_UP

```

```

%left IMPLIES
%left AND OR XOR
%left EQUALITY NOT_EQUALITY
%left LESS LESS_EQUAL GREATOR GREATOR_EQUAL
%left IF THEN ELSE ENDIF
%left PLUS MINUS
%left MULTIPLY DIVIDE DIV MOD
%left NOT
%left DOT ACCESSOR
%left AT
%nonassoc HIGHPREC

```

D.5 Context Declaration

```

%start packageDeclaration
%%
packageDeclaration
: PACKAGE pathName contextDecls ENDPACKAGE
{
    Package p = new Package($2 as PathName, $3 as ArrayList);
    Root.RootPackage = p;
}
| PACKAGE ENDPACKAGE
{
    ErrorHandler.Warning(1001, "Empty package found");
}
| PACKAGE
{
    ErrorHandler.Error(14, "Missing endpackage");
}
contextDecls
{
    Root.ContextDecls = $1 as ArrayList;
}
/* Empty */
{
    ErrorHandler.Warning(1000, "Empty OCL expression found");
}
EOF
{
}
;

contextDecls
: contextDecl contextDecls
{
    ArrayList ar = $2 as ArrayList;
    ar.Insert(0, $1 as ContextDecl);
    $$ = ar;
}
| contextDecl
{
    ArrayList ar = new ArrayList();
    ar.Add($1 as ContextDecl);
    $$ = ar;
}

```

```

;

contextDecl
: attrOrAssocContext { $$ = $1 as ContextDecl; }
| classifierContextDecl { $$ = $1 as ContextDecl; }
| operationContextDecl { $$ = $1 as ContextDecl; }
;

attrOrAssocContext
: CONTEXT pathName COLON type
{
    current_context = new AssocContextDecl($2 as PathName,
        $4 as OclType);
}
initOrDerValue
{
    $$ = current_context;
}
| CONTEXT pathName EXT_GET COLON
{
    current_context = new OperationContextDecl($2 as PathName,
        "get", null);
}
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_GET COLON type
{
    current_context = new OperationContextDecl($2 as PathName,
        "get", $5 as OclType);
}
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_SET COLON
{
    current_context = new OperationContextDecl($2 as PathName,
        "set", null);
}
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_SET COLON type
{
    current_context = new OperationContextDecl($2 as PathName,
        "set", $5 as OclType);
}

```



```

prePostOrBodyDecl
{
    $$ = current_context;
}
;

initOrDerValue
: INIT COLON oclExpression
{
    (current_context as AssocContextDecl).AddInit($3
        as OclExpression);
}
initOrDerValue
| INIT COLON oclExpression
{
    (current_context as AssocContextDecl).AddInit($3
        as OclExpression);
}
| DERIVE COLON oclExpression
{
    (current_context as AssocContextDecl).AddDerive($3
        as OclExpression);
}
initOrDerValue
| DERIVE COLON oclExpression
{
    (current_context as AssocContextDecl).AddDerive($3
        as OclExpression);
}
;

classifierContextDecl
: CONTEXT pathName
{
    current_context = new ClassifierContextDecl($2 as PathName);
}
invOrDef
{
    $$ = current_context;
}
| CONTEXT pathName EXT_INDEXER OPEN_PARENS CLOSE_PARENS EXT_GET COLON
{
    current_context = new OperationContextDecl($2 as PathName,
        "iget", null);
}
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_INDEXER OPEN_PARENS CLOSE_PARENS EXT_GET COLON
type

```

```

    {
        current_context = new OperationContextDecl($2 as PathName,
            "iget", $8 as OclType);
    }
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_INDEXER OPEN_PARENS CLOSE_PARENS EXT_SET COLON
{
    current_context = new OperationContextDecl($2 as PathName,
        "iset", null);
}
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_INDEXER OPEN_PARENS CLOSE_PARENS EXT_SET COLON
    type
{
    current_context = new OperationContextDecl($2 as PathName,
        "iset", $8 as OclType);
}
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_INDEXER OPEN_PARENS parameters CLOSE_PARENS
    EXT_GET COLON
{
    current_context = new OperationContextDecl($2 as PathName,
        "iget", null, $5 as ArrayList);
}
prePostOrBodyDecl
{
    $$ = current_context;
}
| CONTEXT pathName EXT_INDEXER OPEN_PARENS parameters CLOSE_PARENS
    EXT_GET COLON type
{
    current_context = new OperationContextDecl($2 as PathName,
        "iget", $9 as OclType, $5 as ArrayList);
}
prePostOrBodyDecl
{
    $$ = current_context;
}

```

```

| CONTEXT pathName EXT_INDEXER OPEN_PARENS parameters CLOSE_PARENS
  EXT_SET COLON
{
  current_context = new OperationContextDecl($2 as PathName,
    "iset", null, $5 as ArrayList);
}
prePostOrBodyDecl
{
  $$ = current_context;
}
| CONTEXT pathName EXT_INDEXER OPEN_PARENS parameters CLOSE_PARENS
  EXT_SET COLON type
{
  current_context = new OperationContextDecl($2 as PathName,
    "iset", $9 as OclType, $5 as ArrayList);
}
prePostOrBodyDecl
{
  $$ = current_context;
}
;
invOrDef
: INV COLON oclExpression
{
  Inv i = new Inv(null, $3 as OclExpression);
  (current_context as ClassifierContextDecl).AddInv(i);
}
| INV simpleName COLON oclExpression
{
  Inv i = new Inv($2 as SimpleName, $4 as OclExpression);
  (current_context as ClassifierContextDecl).AddInv(i);
}
| INV COLON oclExpression
{
  Inv i = new Inv(null, $3 as OclExpression);
  (current_context as ClassifierContextDecl).AddInv(i);
}
invOrDef
| INV simpleName COLON oclExpression
{
  Inv i = new Inv($2 as SimpleName, $4 as OclExpression);
  (current_context as ClassifierContextDecl).AddInv(i);
}
invOrDef
| DEF COLON defExpression
{
  (current_context as ClassifierContextDecl).AddDef($3 as Def);
}
| DEF simpleName COLON defExpression
{
  Def dx = $4 as Def;
  Def d = new Def($2 as SimpleName, dx.Exp, null, dx.OpPathName,

```

```

        dx.OpType, dx.OpParameters);
    (current_context as ClassifierContextDecl).AddDef(d);
}
| DEF COLON defExpression
{
    (current_context as ClassifierContextDecl).AddDef($3 as Def);
}
invOrDef
| DEF simpleName COLON defExpression
{
    Def dx = $4 as Def;
    Def d = new Def($2 as SimpleName, dx.Exp, dx.Variable,
        dx.OpPathName, dx.OpType, dx.OpParameters);
    (current_context as ClassifierContextDecl).AddDef(d);
}
invOrDef
;
defExpression
: simpleName COLON type EQUALITY oclExpression
{
    VariableDecl vd = new VariableDecl($1 as SimpleName,
        $3 as OclType, null);
    $$ = new Def(null, $5 as OclExpression, vd, null, null, null);
}
| pathName OPEN_PARENS CLOSE_PARENS COLON EQUALITY oclExpression
{
    $$ = new Def(null, $6 as OclExpression, null, $1 as PathName,
        null, null);
}
| pathName OPEN_PARENS parameters CLOSE_PARENS COLON
    EQUALITY oclExpression
{
    $$ = new Def(null, $7 as OclExpression, null, $1 as PathName,
        null, $3 as ArrayList);
}
| pathName OPEN_PARENS parameters CLOSE_PARENS COLON type
    EQUALITY oclExpression
{
    $$ = new Def(null, $8 as OclExpression, null, $1 as PathName,
        $6 as OclType, $3 as ArrayList);
}
| pathName OPEN_PARENS CLOSE_PARENS COLON type EQUALITY oclExpression
{
    $$ = new Def(null, $7 as OclExpression, null, $1 as PathName,
        $5 as OclType, null);
}
}

```

```

;
operationContextDecl
: operation
{
    current_context = $1 as OperationContextDecl;
}
prePostOrBodyDecl
{
    $$ = current_context;
}
;
prePostOrBodyDecl
: PRE COLON oclExpression
{
    Pre p = new Pre(null, $3 as OclExpression);
    (current_context as OperationContextDecl).AddPre(p);
}
prePostOrBodyDecl
| PRE simpleName COLON oclExpression
{
    Pre p = new Pre($2 as SimpleName, $4 as OclExpression);
    (current_context as OperationContextDecl).AddPre(p);
}
prePostOrBodyDecl
| PRE COLON oclExpression
{
    Pre p = new Pre(null, $3 as OclExpression);
    (current_context as OperationContextDecl).AddPre(p);
}
| PRE simpleName COLON oclExpression
{
    Pre p = new Pre($2 as SimpleName, $4 as OclExpression);
    (current_context as OperationContextDecl).AddPre(p);
}
| POST COLON oclExpression
{
    Post p = new Post(null, $3 as OclExpression);
    (current_context as OperationContextDecl).AddPost(p);
}
prePostOrBodyDecl
| POST simpleName COLON oclExpression
{
    Post p = new Post($2 as SimpleName, $4 as OclExpression);
    (current_context as OperationContextDecl).AddPost(p);
}
prePostOrBodyDecl
| POST COLON oclExpression
{
    Post p = new Post(null, $3 as OclExpression);
    (current_context as OperationContextDecl).AddPost(p);
}
| POST simpleName COLON oclExpression
{
    Post p = new Post($2 as SimpleName, $4 as OclExpression);
    (current_context as OperationContextDecl).AddPost(p);
}
| BODY COLON oclExpression
{
    Body b = new Body(null, $3 as OclExpression);

```

```

        (current_context as OperationContextDecl).AddBody(b);
    }
prePostOrBodyDecl
| BODY simpleName COLON oclExpression
{
    Body b = new Body($2 as SimpleName, $4 as OclExpression);
    (current_context as OperationContextDecl).AddBody(b);
}
prePostOrBodyDecl
| BODY COLON oclExpression
{
    Body b = new Body(null, $3 as OclExpression);
    (current_context as OperationContextDecl).AddBody(b);
}
| BODY simpleName COLON oclExpression
{
    Body b = new Body($2 as SimpleName, $4 as OclExpression);
    (current_context as OperationContextDecl).AddBody(b);
}
;

operation
: CONTEXT pathName OPEN_PARENS CLOSE_PARENS COLON
{
    $$ = new OperationContextDecl($2 as PathName, "", null);
}
| CONTEXT pathName OPEN_PARENS parameters CLOSE_PARENS COLON
{
    $$ = new OperationContextDecl($2 as PathName, $4 as ArrayList,
        null);
}
| CONTEXT pathName OPEN_PARENS parameters CLOSE_PARENS COLON type
{
    $$ = new OperationContextDecl($2 as PathName, $4 as ArrayList,
        $7 as OclType);
}
| CONTEXT pathName OPEN_PARENS CLOSE_PARENS COLON type
{
    $$ = new OperationContextDecl($2 as PathName, "", $6 as OclType);
}
| CONTEXT
{
    ErrorHandler.Error(15, "Context specified without context name");
}
;

parameters
: variableDeclaration COMMA parameters
{
    ArrayList ar = $3 as ArrayList;
    ar.Insert(0, $1 as VariableDecl);
    $$ = ar;
}
| variableDeclaration
{
    ArrayList ar = new ArrayList();
    ar.Add($1 as VariableDecl);
}

```

```

    $$ = ar;
  }
;
pathName
: simpleName SCOPE pathName
{
  PathName pn = $3 as PathName;
  pn.List.Insert(0, (SimpleName)$1);
  $$ = pn;
}
| simpleName SCOPE
{
  ErrorHandler.Error(13, "Cannot terminate a path name with '::'");
}
| simpleName
{
  ArrayList ar = new ArrayList();
  ar.Add((SimpleName)$1);
  $$ = new PathName(ar);
}

simpleName
: IDENTIFIER
{
  $$ = new SimpleName((string)$1);
}
;

variableDeclaration
: simpleName
{
  $$ = new VariableDecl($1 as SimpleName, null, null);
}
| simpleName COLON type
{
  $$ = new VariableDecl($1 as SimpleName, $3 as OclType, null);
}
| simpleName COLON type EQUALITY oclExpression
{
  $$ = new VariableDecl($1 as SimpleName, $3 as OclType,
    $5 as OclExpression);
}
| simpleName EQUALITY oclExpression
{
  $$ = new VariableDecl($1 as SimpleName, null,
    $3 as OclExpression);
}
;

type
: pathName
{
  $$ = new SimpleType($1 as PathName);
}
| collectionType
{
  $$ = $1 as CollectionType;
}

```

```

    }
    | tupleType
    {
        $$ = $1 as TupleType;
    }
    ;

collectionType
: collectionTypeIdentifier OPEN_PARENS type CLOSE_PARENS
{
    $$ = new CollectionType((CollectionTypes)$1, $3 as OclType);
}
| collectionTypeIdentifier OPEN_PARENS type
{
    ErrorHandler.Error(16, "Missing ')' on collection type");
}
| collectionTypeIdentifier OPEN_PARENS CLOSE_PARENS
{
    ErrorHandler.Error(17, "Missing 'type' on collection type");
}
| collectionTypeIdentifier
{
    ErrorHandler.Error(18, "Missing '(' 'type' ')' for collection
        type");
}
;

collectionTypeIdentifier
: SET { $$ = CollectionTypes.Set; }
| BAG { $$ = CollectionTypes.Bag; }
| SEQUENCE { $$ = CollectionTypes.Sequence; }
| ORDEREDSET { $$ = CollectionTypes.OrderedSet; }
| COLLECTION { $$ = CollectionTypes.Collection; }
;

tupleType
: TUPLETYPE OPEN_PARENS CLOSE_PARENS
{
    $$ = new TupleType(null);
}
| TUPLETYPE OPEN_PARENS parameters CLOSE_PARENS
{
    $$ = new TupleType($3 as ArrayList);
}
| TUPLETYPE OPEN_PARENS
{
    ErrorHandler.Error(19, "Missing ')' on TupleType definition");
}
| TUPLETYPE OPEN_PARENS parameters
{
    ErrorHandler.Error(19, "Missing ')' on TupleType definition");
}
| TUPLE OPEN_PARENS CLOSE_PARENS
{
    $$ = new TupleType(null);
}
| TUPLE OPEN_PARENS parameters CLOSE_PARENS
{
    $$ = new TupleType($3 as ArrayList);
}

```



```

| TUPLE_OPEN_PARENS
{
  ErrorHandler.Error(19, "Missing ')' on TupleType definition");
}
| TUPLE_OPEN_PARENS parameters
{
  ErrorHandler.Error(19, "Missing ')' on TupleType definition");
}
| TUPLETYPE
{
  ErrorHandler.Error(20, "Invalid use of TupleType keyword");
}
;

```

D.6 Expressions

```

oclExpression
: propertyCallExp { $$ = $1 as OclExpression; }
| variableExp { $$ = $1 as OclExpression; }
| literalExp { $$ = $1 as OclExpression; }
| letExp { $$ = $1 as OclExpression; }
| oclMessageExp { $$ = $1 as OclExpression; }
| ifExp { $$ = $1 as OclExpression; }
;

variableExp
: pathName
{
  $$ = new VariableExp($1 as PathName);
}
;

literalExp
: collectionLiteralExp { $$ = $1 as LiteralExp; }
| tupleLiteralExp { $$ = $1 as LiteralExp; }
| primitiveLiteralExp { $$ = $1 as LiteralExp; }
;

collectionLiteralExp
: collectionTypeIdentifier OPEN_BRACE CLOSE_BRACE
{
  $$ = new CollectionLiteralExp((CollectionTypes)$1);
}
| collectionTypeIdentifier OPEN_BRACE collectionLiteralParts
  CLOSE_BRACE
{
  CollectionLiteralExp cl =
    new CollectionLiteralExp((CollectionTypes)$1);

  ArrayList ar = $3 as ArrayList;
  foreach(CollectionPart p in ar)
    cl.AddPart(p);
  $$ = cl;
}
| collectionTypeIdentifier OPEN_BRACE
{

```

```

        ErrorHandler.Error(21, "Missing '}' on collection type");
    }
;
collectionLiteralParts
: collectionLiteralPart COMMA collectionLiteralParts
{
    ArrayList ar = $3 as ArrayList;
    ar.Insert(0, $1 as CollectionPart);
    $$ = ar;
}
| collectionLiteralPart
{
    ArrayList ar = new ArrayList();
    ar.Add($1 as CollectionPart);
    $$ = ar;
}
;
collectionLiteralPart
: oclExpression DOT_DOT oclExpression
{
    $$ = new CollectionPart($1 as OclExpression, $3 as OclExpression);
}
| oclExpression
{
    $$ = new CollectionPart($1 as OclExpression, null);
}
;
tupleLiteralExp
: TUPLE OPEN_BRACE parameters CLOSE_BRACE
{
    $$ = new TupleLiteralExp($3 as ArrayList);
}
| TUPLE OPEN_BRACE CLOSE_BRACE
{
    ErrorHandler.Error(22, "Missing parameters for Tuple");
}
| TUPLE OPEN_BRACE parameters
{
    ErrorHandler.Error(23, "Missing '}' for Tuple");
}
| TUPLE
{
    ErrorHandler.Error(24, "Invalid use of Tuple keyword");
}
;
primitiveLiteralExp
: integerLiteralExp { $$ = $1 as PrimitiveLiteralExp; }
| realLiteralExp { $$ = $1 as PrimitiveLiteralExp; }
| stringLiteralExp { $$ = $1 as PrimitiveLiteralExp; }
| booleanLiteralExp { $$ = $1 as PrimitiveLiteralExp; }
;
integerLiteralExp
: LITERAL_INTEGER
{
    object v = lexer.Value;
    $$ = new IntegerLiteralExp(new Integer((int)v));
}
;
realLiteralExp

```

```

      : LITERAL_REAL
      {
        object v = lexer.Value;
        $$ = new RealLiteralExp(new Real((double)v)); }
      ;
stringLiteralExp
  : LITERAL_STRING
  {
    object v = lexer.Value;
    $$ = new StringLiteralExp(new OclString((string)v));
  }
  ;
booleanLiteralExp
  : TRUE { $$ = new BooleanLiteralExp(new OclBoolean(true)); }
  | FALSE { $$ = new BooleanLiteralExp(new OclBoolean(false)); }
  ;
ifExp
  : IF oclExpression THEN oclExpression ELSE oclExpression ENDIF
  {
    $$ = new IfExp($2 as OclExpression, $4 as OclExpression,
      $6 as OclExpression);
  }
  ;
letExp
  : LET variableDeclaration
  {
    current_expression = new LetExp();
    (current_expression as LetExp).AddVariable($2 as VariableDecl);
  }
  letExpSub
  {
    $$ = current_expression;
  }
  ;
letExpSub
  : COMMA variableDeclaration
  {
    (current_expression as LetExp).AddVariable($1 as VariableDecl);
  }
  letExpSub
  | IN oclExpression
  {
    (current_expression as LetExp).InExp = $2 as OclExpression;
  }
  ;
oclMessageExp
  : oclExpression UP_UP simpleName OPEN_PARENS CLOSE_PARENS
  {
    $$ = new OclMessageExp($1 as OclExpression,
      MessageOperators.Message, $3 as SimpleName, null);
  }
  | oclExpression UP_UP simpleName OPEN_PARENS oclMessageArguments
    CLOSE_PARENS
  {

```

```

    $$ = new OclMessageExp($1 as OclExpression,
        MessageOperators.Message, $3 as SimpleName, $5 as ArrayList);
}
| oclExpression UP SimpleName OPEN_PARENS CLOSE_PARENS
{
    $$ = new OclMessageExp($1 as OclExpression,
        MessageOperators.IsSent, $3 as SimpleName, null);
}
| oclExpression UP SimpleName OPEN_PARENS oclMessageArguments
    CLOSE_PARENS
{
    $$ = new OclMessageExp($1 as OclExpression,
        MessageOperators.IsSent, $3 as SimpleName, $5 as ArrayList);
}
;
oclMessageArguments
: oclMessageArg COMMA oclMessageArguments
{
    ArrayList ar = $3 as ArrayList;
    ar.Insert(0, $1 as OclMessageArg);
    $$ = ar;
}
| oclMessageArg
{
    ArrayList ar = new ArrayList();
    ar.Add($1 as OclMessageArg);
    $$ = ar;
}
;
oclMessageArg
: QUESTION COLON type
{
    $$ = new OclMessageArg(null, $3 as OclType);
}
| QUESTION
{
    $$ = new OclMessageArg(null, null);
}
| oclExpression
{
    $$ = new OclMessageArg($1 as OclExpression, null);
}
;
propertyCallExp
: loopExp { $$ = $1 as LoopExp; }
| modelPropertyCallExp { $$ = $1 as PropertyCallExp; }
;
loopExp
: iteratorExp { $$ = $1 as LoopExp; }
| iterateExp { $$ = $1 as LoopExp; }
;
iterateExp
: oclExpression ACCESSOR ITERATE OPEN_PARENS

```

```

    {
        current_expression = new IterateExp($1 as OclExpression);
    }
iterateExpSub CLOSE_PARENS
{
    $$ = current_expression as IterateExp;
}
;
iterateExpSub
: variableDeclaration SEMICOLON variableDeclaration PIPE
  oclExpression
{
    (current_expression as IterateExp).AddVariable($1 as
        VariableDecl);
    (current_expression as IterateExp).AddVariable($3 as
        VariableDecl);
    (current_expression as IterateExp).BodyExp = $5 as
        OclExpression;
}
;
iteratorExp
: oclExpression ACCESSOR simpleName OPEN_PARENS variableDeclaration
  SEMICOLON variableDeclaration PIPE oclExpression CLOSE_PARENS
{
    IteratorExp ie = new IteratorExp($1 as OclExpression,
        $3 as SimpleName, null);
    ie.AddVariable($5 as VariableDecl);
    ie.AddVariable($7 as VariableDecl);
    ie.BodyExp = $9 as OclExpression;
    $$ = ie;
}
| oclExpression ACCESSOR simpleName OPEN_PARENS variableDeclaration
  PIPE oclExpression CLOSE_PARENS
{
    IteratorExp ie = new IteratorExp($1 as OclExpression,
        $3 as SimpleName, null);
    ie.AddVariable($5 as VariableDecl);
    ie.BodyExp = $7 as OclExpression;
    $$ = ie;
}
| oclExpression ACCESSOR simpleName OPEN_PARENS oclExpression
  CLOSE_PARENS
{
    IteratorExp ie = new IteratorExp($1 as OclExpression,
        $3 as SimpleName, null);

```

```

        ie.BodyExp = $5 as OclExpression;
        $$ = ie;
    }
    ;
arguments
: oclExpression COMMA arguments
{
    ArrayList ar = $3 as ArrayList;
    ar.Insert(0, new Argument($1 as OclExpression));
    $$ = ar;
}
| oclExpression
{
    ArrayList ar = new ArrayList();
    ar.Add(new Argument($1 as OclExpression));
    $$ = ar;
}
;
modelPropertyCallExp
: operationCallExp { $$ = $1 as ModelPropertyCallExp; }
| attributeCallExp { $$ = $1 as ModelPropertyCallExp; }
| navigationCallExp { $$ = $1 as ModelPropertyCallExp; }
;
operationCallExp
: oclExpression ACCESSOR simpleName OPEN_PARENS CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression, $3 as SimpleName,
        null, null, null, false, true);
}
| oclExpression ACCESSOR simpleName OPEN_PARENS arguments CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression, $3 as SimpleName,
        null, $5 as ArrayList, null, false, true);
}
| OPEN_PARENS oclExpression CLOSE_PARENS
{
    $$ = new OperationCallExp($2 as OclExpression,
        Operators.BRACKET, null);
}
| oclExpression DOT simpleName OPEN_PARENS CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression, $3 as SimpleName,
        null, null, null, false, false);
}
| oclExpression DOT simpleName OPEN_PARENS arguments CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression,
        $3 as SimpleName, null, $5 as ArrayList, null, false, false);
}
| simpleName OPEN_PARENS CLOSE_PARENS
{

```

```

    $$ = new OperationCallExp(null, $1 as SimpleName,
        null, null, null, false, false);
}
| simpleName OPEN_PARENS arguments CLOSE_PARENS
{
    $$ = new OperationCallExp(null, $1 as SimpleName, null,
        $3 as ArrayList, null, false, false);
}
| oclExpression DOT simpleName isMarkedPre OPEN_PARENS CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression,
        $3 as SimpleName, null, null, null, true, false);
}
| oclExpression DOT simpleName isMarkedPre OPEN_PARENS arguments
    CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression,
        $3 as SimpleName, null, $6 as ArrayList, null, true, false);
}
| simpleName isMarkedPre OPEN_PARENS CLOSE_PARENS
{
    $$ = new OperationCallExp(null, $1 as SimpleName,
        null, null, null, true, false);
}
| simpleName isMarkedPre OPEN_PARENS arguments CLOSE_PARENS
{
    $$ = new OperationCallExp(null, $1 as SimpleName, null,
        $4 as ArrayList, null, true, false);
}
| pathName OPEN_PARENS CLOSE_PARENS
{
    $$ = new OperationCallExp(null, null, null, null,
        $1 as PathName, false, false);
}
| pathName OPEN_PARENS arguments CLOSE_PARENS
{
    $$ = new OperationCallExp(null, null, null, $3 as ArrayList,
        $1 as PathName, false, false);
}
| oclExpression MULTIPLY oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.MULTIPLY, $3 as OclExpression);
}

```

```

| oclExpression DIVIDE oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.DIVIDE, $3 as OclExpression);
}
| oclExpression DIV oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.DIV, $3 as OclExpression);
}
| oclExpression MOD oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.MOD, $3 as OclExpression);
}
| oclExpression PLUS oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.ADD, $3 as OclExpression);
}
| oclExpression MINUS oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.SUBTRACT, $3 as OclExpression);
}
| oclExpression GREATOR oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.GREATOR, $3 as OclExpression);
}
| oclExpression LESS oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.LESS, $3 as OclExpression);
}
| oclExpression GREATOR_EQUAL oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.GREATOR_EQUAL, $3 as OclExpression);
}
| oclExpression LESS_EQUAL oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.LESS_EQUAL, $3 as OclExpression);
}

```



```

| oclExpression EQUALITY oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.EQUALITY, $3 as OclExpression);
}
| oclExpression NOT_EQUALITY oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.NOT_EQUALITY, $3 as OclExpression);
}
| oclExpression AND oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.AND, $3 as OclExpression);
}
| oclExpression OR oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.OR, $3 as OclExpression);
}
| oclExpression XOR oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.XOR, $3 as OclExpression);
}
| oclExpression IMPLIES oclExpression
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.IMPLIES, $3 as OclExpression);
}
| MINUS oclExpression
{
    $$ = new OperationCallExp(Operators.NEGATION,
        $2 as OclExpression);
}
| NOT oclExpression
{
    $$ = new OperationCallExp(Operators.NOT, $2 as OclExpression);
}
| oclExpression DOT AND OPEN_PARENS oclExpression CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression,
        Operators.AND, $5 as OclExpression);
}
| oclExpression DOT OR OPEN_PARENS oclExpression CLOSE_PARENS
{
    $$ = new OperationCallExp($1 as OclExpression,

```

```

        Operators.OR, $5 as OclExpression);
    }
    | oclExpression DOT XOR OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.XOR, $5 as OclExpression);
    }
    | oclExpression DOT IMPLIES OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.IMPLIES, $5 as OclExpression);
    }
    | oclExpression DOT NOT OPEN_PARENS CLOSE_PARENS
    {
        $$ = new OperationCallExp(Operators.NOT, $1 as OclExpression);
    }
    | oclExpression DOT EQUALITY OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.EQUALITY, $5 as OclExpression);
    }
    | oclExpression DOT NOT_EQUALITY OPEN_PARENS oclExpression
        CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.NOT_EQUALITY, $5 as OclExpression);
    }
    | oclExpression DOT MULTIPLY OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.MULTIPLY, $5 as OclExpression);
    }
    | oclExpression DOT DIVIDE OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.DIVIDE, $5 as OclExpression);
    }
    | oclExpression DOT DIV OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.DIV, $5 as OclExpression);
    }
    | oclExpression DOT MOD OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,

```

```

        Operators.MOD, $5 as OclExpression);
    }
    | oclExpression DOT PLUS OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.ADD, $5 as OclExpression);
    }
    | oclExpression DOT MINUS OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.SUBTRACT, $5 as OclExpression);
    }
    | oclExpression DOT GREATOR OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.GREATOR, $5 as OclExpression);
    }
    | oclExpression DOT LESS OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.LESS, $5 as OclExpression);
    }
    | oclExpression DOT GREATOR_EQUAL OPEN_PARENS oclExpression
        CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.GREATOR_EQUAL, $5 as OclExpression);
    }
    | oclExpression DOT LESS_EQUAL OPEN_PARENS oclExpression CLOSE_PARENS
    {
        $$ = new OperationCallExp($1 as OclExpression,
            Operators.LESS_EQUAL, $5 as OclExpression);
    }
    | oclExpression DOT MINUS OPEN_PARENS CLOSE_PARENS
    {
        $$ = new OperationCallExp(Operators.NEGATION, $1 as OclExpression);
    }
    ;
isMarkedPre
: AT PRE
{
}
;
attributeCallExp
: oclExpression DOT simpleName isMarkedPre
{
    $$ = new AttributeCallExp($1 as OclExpression,

```

```

        $3 as SimpleName, null, true);
    }
    | oclExpression DOT SimpleName
    {
        $$ = new AttributeCallExp($1 as OclExpression,
            $3 as SimpleName, null, false);
    }
    | SimpleName isMarkedPre
    {
        $$ = new AttributeCallExp(null, $1 as SimpleName, null, true);
    }
navigationCallExp
: oclExpression DOT SimpleName OPEN_BRACKET arguments
  CLOSE_BRACKET isMarkedPre
{
    $$ = new NavigationCallExp($1 as OclExpression,
        $3 as SimpleName, $5 as ArrayList, true);
}
| SimpleName OPEN_BRACKET arguments CLOSE_BRACKET isMarkedPre
{
    $$ = new NavigationCallExp(null, $1 as SimpleName,
        $3 as ArrayList, true);
}
| SimpleName OPEN_BRACKET arguments CLOSE_BRACKET
{
    $$ = new NavigationCallExp(null, $1 as SimpleName,
        $3 as ArrayList, false);
}
}
;
%%

```

D.7 Grammar Footer

```

OCLTokenizer lexer;
OCLRoot root = new OCLRoot();
public OCLTokenizer Lexer
{
    get
    {
        return lexer;
    }
}
public OCLRoot Root
{
    get { return root; }
}
public OCLParser(String s)

```

```
{
    lexer = new OCLTokenizer(s);
}
public void parse()
{
    try
    {
        yyparse(lexer);
    }
    catch(Exception e)
    {
        Console.WriteLine(e.StackTrace);
        Console.WriteLine(e.Message);
        throw e;
    }
}
void CheckToken(int error, int yyToken, string msg)
{
    if(yyToken >= Token.FIRST_KEYWORD && yyToken <= Token.LAST_KEYWORD)
    {
        ErrorHandler.Error(error, String.Format("{0}: '{1}' is a keyword",
            msg, yyName [yyToken].ToLower ()));
        return;
    }
    ErrorHandler.Error(error, msg);
}
void CheckIdentifierToken(int yyToken)
{
    CheckToken(10, yyToken, "Identifier expected");
}
}
```

Appendix E

OCL Compiler Error Codes and Warnings

This appendix contains all of the error and warning codes that the specialized compiler can generate with respect to OCL expressions. The list contains both syntactic and semantic errors. Following the errors, the warning list is displayed.

E.1 Errors

Code	Message
1	The OCL 2.0 Standard Library cannot be used by user defined code.
2	Invalid use of the OCL keyword.
3	Numeric constant too long.
4	Integral constant is too large
5	Floating-point constant is outside the range of the type 'Real'
6	Unrecognized escape sequence in X
7	Newline in constant
8	Unterminated string literal
9	Identifier too long (limit is 512 chars)
10	Identifier expected
11	Source file 'X' specified multiple times
12	Parse error on OCL block. Stopping main compile.

13	Cannot terminate a path name with '::'
14	Missing endpackage
15	Context specified without context name
16	Missing ')' on collection type
17	Missing 'type' on collection type
18	Missing '(' 'type' ')' for collection type
19	Missing ')' on TupleType definition
20	Invalid use of TupleType keyword
21	Missing '}' on collection type
22	Missing parameters for Tuple
23	Missing '}' for Tuple
24	Invalid use of Tuple keyword
25	Missing endif
26	Missing then
27	Missing else
28	Unknown context declaration type
29	Cannot find context name: 'X'
30	Association context declarations cannot be applied to classifiers
31	Invalid collection type
32	Undefined type: 'X'
33	Cannot create attribute: 'X' for derive rule
34	More then one derive rule of the same name (X) but different types.
35	Undefined classifier: 'X'
36	Attribute 'X' is already defined. Cannot create attribute via def rule
37	Property 'X' is already defined. Cannot create attribute via def rule
38	Variable type must be specified when using def rules
39	Operation 'X' already exists and cannot be defined via a def rule
40	Parameter types must be specified on def rules
41	Operations defined via a def rule must specify a return type
42	Cannot create operation 'X' for def rule
43	Parameter names must be specified on def rules
44	Bad operation name for 'X'
45	Undefined classifier in: 'X'
46	Parameter types must be specified on operation context definitions
47	The operation 'X' already exists and cannot be specified by a body expression
48	Operations used in a context expression must specify a return type
49	Return type mismatch in 'X'
50	Undefined operation: 'X'
51	Operations defined via a body rule must specify a return type
52	Parameter types must be specified on body expressions
53	Parameter names must be specified on body expressions
54	Cannot create operation 'X' for body expression
55	More than one operation matches OCL context: 'X'
56	Constructors can not have a return type
57	Cannot create constructor 'X' for body expression

58	More than one constructor matches OCL context: 'X'
59	init and derive rules can not be specified on Interface elements
60	inv and def expressions can not be specified on Interface elements
61	C# properties can only be used if the Treat C# Properties as Methods OCL compiler option is selected
62	More than one property matches OCL context: 'X'
63	Property type mismatch in 'X'
64	The property 'X' already exists and cannot be specified by a body expression
65	The property 'X' does not contain a get accessor
66	The property 'X' does not contain a set accessor
67	Undefined property: 'X'
68	Properties defined via a body expression must specify the return type
69	Cannot create property 'X' for body expression
70	Body rules can not be specified on properties in Interfaces
71	Field 'X' type does not match init rule type
72	Field 'X' has already been assigned and can not be assigned via an init rule
73	Init rule for field 'X' does not specify a valid type
74	More than one OCL type for init rule. Internal OCL compiler error
75	OCL init rule for field 'X' context type does not match actual init rule type
76	Field 'X' can not have more than one OCL init rule
77	Cannot map enumeration 'X' to a C# type
78	Enum literal expressions should not exist. Internal compiler error
79	Collection is abstract and cannot be used as a constant
80	Undefined collection type. Internal compiler error
81	Invalid use of '..' operator for collection constant (expecting two Integer types)
82	An element of a collection constant could not be converted to a constant
83	Undefined collection element type. Internal compiler error
84	Collection element 'X' does not match the collection type
85	Sets and OrderedSets cannot contain duplicate elements
86	Found internal collection but item type was not a collection type
87	Invalid collection type for internal collection (must be concrete collection type)
88	Invalid collection type for collection (must be collection type)
89	Cannot resolve tuple literal value into a constant: 'X'
90	Unknown type from new expression. Can not create. Internal compiler error
91	Unknown element type in collection. Can not create item. Internal compiler error
92	Found tuple when looking for non tuple type
93	Tuple literal does not match the given tuple type
94	Tuple already contains a definition for name: 'X'
95	Can not find a type conversion for C# constant
96	Cannot convert init expression into constant. Init expressions must resolve into a constant for use
97	Variable 'X' cannot be mapped to a constant
98	Constants can not be defined using derive rules
99	Constant 'X' can not have more than one OCL init rule
100	Init rule for constant 'X' does not specify a valid type
101	OCL init rule for constant 'X' context type does not match actual init rule type

102	Constant 'X' type does not match init rule type
103	Init rule for constant: 'X' must result in a true constant (Integer, Real, Boolean, or String)
104	Constant 'X' type does not match init rule type
105	Constants defined via an OCL init rule can not be used to define other constants
106	The '..' operator requires that the first integer is smaller than the second integer
107	An Integer or Real type can only be added to a Real type ('Real + X')
108	Unknown binary operator for type Real
109	An Integer or Real type can only be subtracted from a Real type ('Real - X')
110	An Integer or Real type can only be multiplied from a Real type ('Real * X')
111	Division by zero in constant expression
112	An Integer or Real type can only be divided by a Real type ('Real / X')
113	An Integer or Real type can only be used in a logical expression with a Real type ('Real < X')
114	An Integer or Real type can only be used in a logical expression with a Real type ('Real > X')
115	An Integer or Real type can only be used in a logical expression with a Real type ('Real <= X')
116	An Integer or Real type can only be used in a logical expression with a Real type ('Real >= X')
117	The 'and' operator can not be applied to Real types
118	The 'div' operator can not be applied to Real types
119	An Integer or Real type can only be used in a logical expression with a Real type ('Real = X')
120	The 'implies' operator can not be applied to Real types
121	The 'mod' operator can not be applied to Real types
122	An Integer or Real type can only be used in a logical expression with a Real type ('Real != X')
123	The 'or' operator can not be applied to Real types
124	The 'xor' operator can not be applied to Real types
125	The 'not' operator can not be applied to Real types
126	Unknown unary operator for type Real
127	The 'not' operator can not be applied to Integer types
128	Unknown unary operator for type Integer
129	An Integer type can only be added to an Integer type ('Integer + X')
130	An Integer type can only be subtracted from an Integer type ('Integer - X')
131	An Integer type can only be multiplied with an Integer type ('Integer * X')
132	An Integer type can only be divided by an Integer ('Integer / X')
133	An Integer or Real type can only be used in a logical expression with an Integer type ('Integer < X')
134	An Integer or Real type can only be used in a logical expression with an Integer type ('Integer > X')
135	An Integer or Real type can only be used in a logical expression with an Integer type ('Integer <= X')
136	An Integer or Real type can only be used in a logical expression with an Integer type ('Integer >= X')
137	The 'and' operator can not be applied to Integer types
138	An Integer type can only be used in a 'div' expression with an Integer type ('Integer.div(X)')
139	An Integer or Real type can only be used in a logical expression with an Integer type ('Integer = X')
140	The 'implies' operator can not be applied to Integer types
141	An Integer type can only be used in a 'mod' expression with an Integer type ('Integer.mod(X)')
142	An Integer or Real type can only be used in a logical expression with an Integer type ('Integer != X')
143	The 'or' operator can not be applied to Integer types
144	The 'xor' operator can not be applied to Integer types

145	Unknown binary operator for type Integer
146	The '-' operator can not be applied to String types
147	The 'not' operator can not be applied to String types
148	Unknown unary operator for type String
149	The '+' operator can not be applied to String types
150	The '*' operator can not be applied to String types
151	The '/' operator can not be applied to String types
152	The '<' operator can not be applied to String types
153	The '>' operator can not be applied to String types
154	The '<=' operator can not be applied to String types
155	The '>=' operator can not be applied to String types
156	The 'and' operator can not be applied to String types
157	The 'div' operator can not be applied to String types
158	A String type can only be used in a logical expression with a String type ('String = X')
159	The 'implies' operator can not be applied to String types
160	The 'mod' operator can not be applied to String types
161	A String type can only be used in a logical expression with a String type ('String != X')
162	The 'or' operator can not be applied to String types
163	The 'xor' operator can not be applied to String types
164	Unknown binary operator for type String
165	The '-' operator can not be applied to Boolean types
166	Unknown unary operator for type Boolean
167	The '+' operator can not be applied to Boolean types
168	The '-' operator can not be applied to Boolean types
169	The '*' operator can not be applied to Boolean types
170	The '/' operator can not be applied to Boolean types
171	The '<' operator can not be applied to Boolean types
172	The '>' operator can not be applied to Boolean types
173	The '<=' operator can not be applied to Boolean types
174	The '>=' operator can not be applied to Boolean types
175	A Boolean type can only be anded with another Boolean type
176	The 'div' operator can not be applied to Boolean types
177	A Boolean type can only be used in a logical expression with a Boolean type ('Boolean = X')
178	A Boolean type can only be implied with another Boolean type
179	The 'mod' operator can not be applied to Boolean types
180	A Boolean type can only be used in a logical expression with a Boolean type ('Boolean != X')
181	A Boolean type can only be ored with another Boolean type
182	A Boolean type can only be xored with another Boolean type
183	Unknown binary operator for type Boolean
184	The abs() operation does not take any parameters
185	The operation 'X' is not defined on Real types
186	The floor() operation does not take any parameters
187	The round() operation does not take any parameters
188	The max(r : Real) operation requires one parameter of type Real or Integer
189	The min(r : Real) operation requires one parameter of type Real or Integer

190	The <code>max(i : Integer)</code> operation requires one parameter of type Integer
191	The <code>min(i : Integer)</code> operation requires one parameter of type Integer
192	The operation 'X' is not defined on Integer types
193	The <code>div(i : Integer)</code> operation requires one parameter of type Integer
194	The <code>mod(i : Integer)</code> operation requires one parameter of type Integer
195	The <code>size()</code> operation does not take any parameters
196	The <code>concat(s : String)</code> operation takes one parameter of type String
197	The operation 'X' is not defined on String types
198	The <code>substring(lower : Integer, upper : Integer)</code> operation takes two parameters of type Integer
199	The <code>toInteger()</code> operation does not take any parameters
200	The <code>toReal()</code> operation does not take any parameters
201	The operation 'X' is not defined on Boolean types
202	The '-' operator can not be applied to Enumeration types
203	The 'not' operator can not be applied to Enumeration types
204	Unknown unary operator for type Enumeration
205	The '+' operator can not be applied to Enumeration types
206	The '-' operator can not be applied to Enumeration types
207	The '*' operator can not be applied to Enumeration types
208	The '/' operator can not be applied to Enumeration types
209	The '<' operator can not be applied to Enumeration types
210	The '>' operator can not be applied to Enumeration types
211	The '<=' operator can not be applied to Enumeration types
212	The '>=' operator can not be applied to Enumeration types
213	The 'and' operator can not be applied to Enumeration types
214	The 'div' operator can not be applied to Enumeration types
215	An Enumeration type can only be used in a logical expression with an Enumeration type ('Enumeration = X')
216	The 'implies' operator can not be applied to Enumeration types
217	The 'mod' operator can not be applied to Enumeration types
218	An Enumeration type can only be used in a logical expression with an Enumeration type ('Enumeration <> X')
219	The 'or' operator can not be applied to Enumeration types
220	The 'xor' operator can not be applied to Enumeration types
221	Unknown binary operator for type Enumeration
222	The operation 'X' is not defined on Enumeration types
223	The operation 'X' is not defined on Collection types
224	The <code>Collection::size()</code> operation does not take any parameters
225	The <code>Collection::includes(object : T)</code> operation requires one parameter of type T
226	The <code>Collection::excludes(object : T)</code> operation requires one parameter of type T
227	The <code>Collection::count(object : T)</code> operation requires one parameter of type T
228	The <code>Collection::includesAll(c2 : Collection(T))</code> operation requires one parameter of type Collection(T)
229	The <code>Collection::excludesAll(c2 : Collection(T))</code> operation requires one parameter of type Collection(T)
230	The <code>Collection::isEmpty()</code> operation does not take any parameters

231	The Collection::notEmpty() operation does not take any parameters
232	The Collection::sum() operation does not take any parameters
233	The Collection::sum() operation requires an element type which supports the '+' operation
234	The OclAny::oclIsNew() operation does not take any parameters
235	The OclAny::oclIsNew() operation can not be used in constant expressions
236	The OclAny::oclIsUndefined() operation does not take any parameters
237	The OclAny::oclAsType(typename : OclType) operation takes one parmater
238	The OclAny::oclAsType(typename : OclType) operation can not convert parameter 'typename' to OclType
239	The OclAny::oclAsType(typename : OclType) operation can not convert parameter 'typename' to a constant type
240	The OclAny::oclIsTypeOf(typename : OclType) operation can not convert parameter 'typename' to OclType
241	The OclAny::oclIsTypeOf(typename : OclType) operation takes one parmater
242	The OclAny::oclIsTypeOf(typename : OclType) operation can not convert parameter 'typename' to OclType
243	The OclAny::oclIsKindOf(typename : OclType) operation can not convert parameter 'typename' to OclType
244	The OclAny::oclIsKindOf(typename : OclType) operation can not convert parameter 'typename' to OclType
245	The OclAny::oclIsKindOf(typename : OclType) operation takes one parmater
246	The OclAny::oclIsInState(statename : OclState) operation takes one parmater
247	The OclAny::allInstances operation is not supported
248	The '-' operator can not be applied to the OclVoid type
249	The 'not' operator can not be applied to the OclVoid type
250	Unknown unary operator for type OclVoid
251	The '+' operator can not be applied to the OclVoid type
252	The '-' operator can not be applied to the OclVoid type
253	The '*' operator can not be applied to the OclVoid type
254	The '/' operator can not be applied to the OclVoid type
255	The '<' operator can not be applied to the OclVoid type
256	The '>' operator can not be applied to the OclVoid type
257	The '<=' operator can not be applied to the OclVoid type
258	The '>=' operator can not be applied to the OclVoid type
259	The 'and' operator can not be applied to the OclVoid type
260	The 'div' operator can not be applied to the OclVoid type
261	The 'implies' operator can not be applied to the OclVoid type
262	The 'mod' operator can not be applied to the OclVoid type
263	The 'or' operator can not be applied to the OclVoid type
264	The 'xor' operator can not be applied to the OclVoid type
265	Unknown binary operator for type OclVoid
266	The operation 'X' is not defined on OclVoid types
267	The Collection::product(c2 : Collection(T2)) operation takes one parameter of type Collection
268	A Collection type can only be used in a logical expression with a Collection type ('Collection = X')
269	Unknown binary operator for type Collection

270	A Collection type can only be used in a logical expression with a Collection type ('Collection != X')
271	A Set type can only be used in a logical expression with a Set type ('Set = X')
272	A Set type can only be used in a logical expression with a Set type ('Set != X')
273	Unknown binary operator for type Set
274	An OrderedSet type can only be used in a logical expression with an Ordered Set type ('OrderedSet = X')
275	An OrderedSet type can only be used in a logical expression with an OrderedSet type ('OrderedSet != X')
276	Unknown binary operator for type OrderedSet
277	A Bag type can only be used in a logical expression with a Bag type ('Bag = X')
278	A Bag type can only be used in a logical expression with a Bag type ('Bag != X')
279	Unknown binary operator for type Bag
280	A Sequence type can only be used in a logical expression with a Sequence type ('Sequence = X')
281	A Sequence type can only be used in a logical expression with a Sequence type ('Sequence != X')
282	Unknown binary operator for type Sequence
283	Unknown binary operator for type Tuple
284	A Set type can only be subtracted from another Set type ('Set - X')
285	The Set::union(s : Set(T)) operation requires one parameter of type Set(T)
286	The collection element type of the parameter must match the collection element type of the object
287	The Set::intersection(s : Set(T)) operation requires one parameter of type Set(T)
288	The parameter type does not match the collection element type
289	The Set::including(object : T) operation requires one parameter of type T
290	The Set::excluding(object : T) operation requires one parameter of type T
291	The Set::symmetricDifference(s : Set(T)) operation requires one parameter of type Set(T)
292	The Set::count(object : T) operation requires one parameter of type T
293	The Set::flatten() operation does not take any parameters
294	The Set::asSet() operation does not take any parameters
295	The Set::asOrderedSet() operation does not take any parameters
296	The Set::asSequence() operation does not take any parameters
297	The Set::asBag() operation does not take any parameters
298	The operation 'X' is not defined on Set types
299	The operation 'X' is not defined on OrderedSet types
300	The operation 'X' is not defined on Sequence types
301	The operation 'X' is not defined on Bag types
302	The OrderedSet::append(object : T) operation requires one parameter of type T
303	The OrderedSet::prepend(object : T) operation requires one parameter of type T
304	Parameter one for OrderedSet::insertAt(index : Integer, object : T) is not an Integer type
305	Parameter two for OrderedSet::insertAt(index : Integer, object : T) is not of type T
306	The OrderedSet::insertAt(index : Integer, object : T) operation requires two parameters
307	Parameter two for OrderedSet::subOrderdSet(lower : Integer, upper : Integer) is not an Integer type
308	Parameter one for OrderedSet::subOrderdSet(lower : Integer, upper : Integer) is not an Integer type
309	The OrderedSet::subOrderdSet(lower : Integer, upper : Integer) operation requires two parameters of type Integer
310	Parameter one for OrderedSet::at(i : Integer) is not an Integer type
311	The OrderedSet::at(i : Integer) operation requires one parameter of type Integer

312	The <code>OrderedSet::indexOf(object : T)</code> operation requires one parameter of type T
313	The <code>OrderedSet::first()</code> operation does not take any parameters
314	The <code>Bag::union(bag : Bag(T))</code> operation requires one parameter of type Bag(T)
315	The <code>Bag::intersection(bag : Bag(T))</code> operation requires one parameter of type Bag(T)
316	The <code>Bag::including(object : T)</code> operation requires one parameter of type T
317	The <code>Bag::excluding(object : T)</code> operation requires one parameter of type T
318	The <code>Bag::count(object : T)</code> operation requires one parameter of type T
319	The <code>Bag::flatten()</code> operation does not take any parameters
320	The <code>Bag::asSet()</code> operation does not take any parameters
321	The <code>Bag::asOrderedSet()</code> operation does not take any parameters
322	The <code>Bag::asSequence()</code> operation does not take any parameters
323	The <code>Bag::asBag()</code> operation does not take any parameters
324	The <code>Sequence::union(s : Sequence(T))</code> operation requires one parameter of type Sequence(T)
325	The <code>Sequence::including(object : T)</code> operation requires one parameter of type T
326	The <code>Sequence::excluding(object : T)</code> operation requires one parameter of type T
327	The <code>Sequence::count(object : T)</code> operation requires one parameter of type T
328	The <code>Sequence::flatten()</code> operation does not take any parameters
329	The <code>Sequence::asSet()</code> operation does not take any parameters
330	The <code>Sequence::asOrderedSet()</code> operation does not take any parameters
331	The <code>Sequence::asSequence()</code> operation does not take any parameters
332	The <code>Sequence::asBag()</code> operation does not take any parameters
333	The <code>Sequence::append(object : T)</code> operation requires one parameter of type T
334	The <code>Sequence::prepend(object : T)</code> operation requires one parameter of type T
335	Parameter two for <code>Sequence::insertAt(index : Integer, object : T)</code> is not of type T
336	Parameter one for <code>Sequence::insertAt(index : Integer, object : T)</code> is not an Integer type
337	The <code>Sequence::insertAt(index : Integer, object : T)</code> operation requires two parameters
338	Parameter two for <code>Sequence::subSequence(lower : Integer, upper : Integer)</code> is not an Integer type
339	Parameter one for <code>Sequence::subSequence(lower : Integer, upper : Integer)</code> is not an Integer type
340	The <code>Sequence::subSequence(lower : Integer, upper : Integer)</code> operation requires two parameters of type Integer
341	Parameter one for <code>Sequence::at(i : Integer)</code> is not an Integer type
342	The <code>Sequence::at(i : Integer)</code> operation requires one parameter of type Integer
343	The <code>Sequence::indexOf(object : T)</code> operation requires one parameter of type T
344	The <code>Sequence::first()</code> operation does not take any parameters
345	The <code>Sequence::last()</code> operation does not take any parameters
346	A Tuple type can only be used in a logical expression with a Tuple type ('Tuple = X')
347	A Tuple type can only be used in a logical expression with a Tuple type ('Tuple <> X')
348	Tuple literal does not contain an attribute named 'X'
349	Derive rule for property 'X' does not specify a valid type
350	More than one OCL type for derive rule. Internal OCL compiler error
351	Cannot convert derive expression into constant. Derive expressions must resolve into a constant for use
352	Property 'X' has already been defined and can not be defined via an derive rule
353	OCL derive rule for property 'X' context type does not match actual derive rule type
354	Property 'X' type does not match derive rule type

355	Cannot find type for attribute 'X'
356	The operation 'X' is not defined on Tuple types
357	Each Set must be composed of the same element type
358	Logical operators may only be applied to collection types that contain the same element types
359	TupleType does not contain an attribute named X
360	The 'self' keyword cannot be resolved into an instance type
361	Body rule for operation 'X' does not specify a valid type
362	More than one OCL type for body rule. Internal OCL compiler error
363	Operation 'X' type does not match body rule type
364	Operation 'X' has already been defined and can not be defined via a body rule
365	OCL body rule for operation 'X' context type does not match actual body rule type
366	Body rule for property 'X' does not specify a valid type
367	Property 'X' type does not match body rule type
368	Property 'X' has already been defined and can not be defined via a body rule
369	OCL body rule for property 'X' context type does not match actual body rule type
370	Body rules can not be used to define a SET accessor, because a SET accessor would have side-effects
371	Def rule for property 'X' does not specify a valid type
372	More than one OCL type for def rule. Internal OCL compiler error
373	Property 'X' type does not match def rule type
374	Property 'X' has already been defined and can not be defined via an def rule
375	OCL def rule for property 'X' context type does not match actual def rule type
376	Operation 'X' type does not match def rule type
377	OCL def rule for operation 'X' context type does not match actual def rule type
378	Operation 'X' type does not match def rule type
379	Cannot resolve pre condition into boolean expression
380	Could not add new variable to 'X', internal compiler error
381	Cannot resolve post condition into boolean expression
382	The result keyword can only be used in the context of post conditions
383	The @pre modifier must be used on post conditions
384	Cannot resolve invariant into boolean expression
385	Cannot locate attribute named: 'X'
386	Cannot specify a pre condition on a get accessor of a property that does not exist
387	Cannot specify a pre condition on a set accessor of a property that does not exist
388	Cannot specify a post condition on a get accessor of a property that does not exist
389	Cannot specify a post condition on a set accessor of a property that does not exist
390	The result keyword can not be used in an operation that has a return type of OclVoid
391	More than one indexer matches OCL context: 'X'
392	Indexer type mismatch in 'X'
393	Body rules can not be specified on indexers in Interfaces
394	The indexer 'X' does not contain a get accessor
395	The indexer 'X' does not contain a set accessor
396	Undefined indexer: 'X'
397	The indexer 'X' already exists and cannot be specified by a body expression
398	Indexers defined via a body expression must specify the return type

399	Cannot create indexer 'X' for body expression
400	Indexer 'X' type does not match body rule type
401	Indexer 'X' has already been defined and can not be defined via a body rule
402	Cannot specify a pre condition on a get accessor of an indexer that does not exist
403	Cannot specify a pre condition on a set accessor of an indexer that does not exist
404	Cannot specify a post condition on a get accessor of an indexer that does not exist
405	Cannot specify a post condition on a set accessor of an indexer that does not exist
406	Cannot resolve interface 'X' for inheritance lookup
407	If statement condition must evaluate to a boolean expression
408	If statement then clause and else clause must evaluate to the same type
409	The oclAsType operation takes one parameter of type Type
410	The oclAsType operation cannot convert type: 'X' to type: 'X'
411	The oclIsTypeOf operation cannot convert type: 'X' to type: 'X'
412	The oclIsKindOf operation cannot convert type: 'X' to type: 'X'
413	The oclIsTypeOf operation takes one parameter of type Type
414	The oclIsKindOf operation takes one parameter of type Type
415	The OclAny::allInstances() operation does not take any parameters
416	The OclAny::oclIsNew() operation can only be used in a post condition
417	The oclIsInState operation cannot convert type: 'X' to type: 'X'
418	The operation 'X' is not defined on type 'X'
419	The operation 'X' does not match the signature of the operation in 'X'
420	Variable promotion for OCL attribute reference 'X' cannot be completed because a variable with the same name already exists at the parent scope
421	Cannot compare an instance of type: 'X' with an instance of type: 'X'
422	Unknown binary operator for type: 'X'
423	Operations not marked with the [Query] attribute can not be called via an OCL expression
424	The -> operator must have a valid left expression
425	Unknown static attribute: 'X' defined on type: 'X'
426	Unknown static attribute: 'X' defined on type: 'X' is write only
427	Iterator variable types must be specified on collection types that are not strongly typed
428	Cannot derive the collection type for this iterator expression
429	Iterator variable 'X' type does not match the collection type
430	Iterator 'X' is not defined on any collection type
431	The body expression for the exists iterator must evaluate to a Boolean expression
432	The body expression for the forAll iterator must evaluate to a Boolean expression
433	The exists iterator only supports a single iterator variable
434	The forAll iterator only supports one or two iterator variables
435	The isUnique iterator only supports a single iterator variable
436	The body expression for the any iterator must evaluate to a Boolean expression
437	The any iterator only supports a single iterator variable
438	The one iterator only supports a single iterator variable
439	The collect iterator only supports a single iterator variable
440	The select iterator only supports a single iterator variable
441	The select iterator is only defined on the Set, Bag, and Sequence collection types
442	The reject iterator only supports a single iterator variable

443	The reject iterator is only defined on the Set, Bag, and Sequence collection types
444	The collectNested iterator only supports a single iterator variable
445	The sortedBy iterator only supports a single iterator variable
446	The sortedBy iterator is only defined on the Set, Bag, and Sequence collection types
447	The Iterate expression requires two variables
448	Iterate variable types must be specified on collection types that are not strongly typed
449	The accumulator variable definition must contain the variable type
450	The accumulator variable definition must contain an initial expression
451	Cannot derive the collection type for this iterate expression
452	Iterator variable 'X' type in an iterate expression does not match the collection type
453	The body expression for the iterate expression must evaluate to the same type as the accumulator variable
454	The accumulator initialization expression does not match the accumulator variable type

E.2 Warnings

Code	Message
1000	Empty OCL expression found
1001	Empty package found
1002	Context specification located on incorrect classifier: 'X'. Moving context to correct classifier: 'X'
1003	Parameter expression ignored in parameter 'X'
1004	Conversion from System::Int16 to Integer (System::Int32)
1005	Conversion from System::Int64 to Integer (System::Int32). Possible loss of data
1006	Conversion from System::UInt16 to Integer (System::Int32). Possible loss of data
1007	Conversion from System::UInt32 to Integer (System::Int32). Possible loss of data
1008	Conversion from System::UInt64 to Integer (System::Int32). Possible loss of data
1009	Conversion from System::Decimal to Real (System::Double). Possible loss of data
1010	Return type not specified in OCL, using return type from C#
1011	Conversion from System::Single to Real (System::Double). Possible loss of data
1012	Conversion from System::Decimal to Integer (System::Int32). Possible loss of data
1013	Conversion from System::Single to Integer (System::Int32). Possible loss of data
1014	At least one class or struct is required to process OCL expressions. OCL will not be processed on enums or delegates
1015	Conversion from System::Byte to Integer (System::Int32)
1016	Conversion from System::Char to Integer (System::Int32)
1017	Conversion from System::SByte to Integer (System::Int32)
1018	Conversion from System::Short to Integer (System::Int32)
1019	Conversion from System::UShort to Integer (System::Int32)

1020	Init rule specified for constant that already contains a value. The expression specified via the init rule will be used to initialize the constant
1021	Weakly typed OCL collection found, using C# information to make the collection strongly typed

Appendix F

Select Test Cases

This appendix contains three of the thirty test cases. The test cases contained in this section apply to iterator expressions, attribute navigation expressions, and type expressions (casting and state). The complete set of test cases is included as part of the implementation. For information relating to the implementation please see [2].

F.1 Iterator Expressions

```
// -----  
// Iterator.cs: Represents OCL testing for Iterator expressions  
// -----  
// Project: C# and OCL Compiler  
// Module: Testing  
// Author: Dave Arnold  
// Version: 1.0  
// -----  
// -----  
// Imports  
// -----  
using System;  
using System.Collections;  
// -----  
// -----  
// DaveArnold.OCLTesting Namespace  
// -----  
namespace DaveArnold.OCLTesting  
{  
    // A Class to test Iterator expressions  
    class IteratorTests
```

```

{
    public class Person
    {
        public int Age = 0;
        [Query]
        public int GetAge()
        {
            return Age;
        }
        public Person(int a)
        {
            Age = a;
        }
    }
    public Set _pSet = Set.EmptySet();
    public Person p1 = new Person(10);
    public Person p2 = new Person(20);
    OCL
    [
        "context IteratorTests::Exists() : OclVoid"
        "pre: Set { 1, 2, 3, 4, 5 }->exists(t | t > 4)"
    ]
    public void Exists()
    {
    }
    OCL
    [
        "context IteratorTests::ForAll() : OclVoid"
        "pre: Set { 1, 2, 3, 4, 5 }->forAll(t | t >= 1)"
    ]
    public void ForAll()
    {
    }
    OCL
    [
        "context IteratorTests::ForAll2() : OclVoid"
        "post: _pSet->forAll(p : Person | p.Age <= 60)"
    ]
    public void ForAll2()
    {
        _pSet._col.Add(new Person(10));
        _pSet._col.Add(new Person(20));
        _pSet._col.Add(new Person(30));
        _pSet._col.Add(new Person(40));
    }
    OCL
    [
        "context IteratorTests::ForAll3() : OclVoid"
        "pre: Set { p1, p2 }->forAll(Age <= 30)"
    ]
    public void ForAll3()
    {
    }
    OCL
    [
        "context IteratorTests::ForAll4() : OclVoid"
        "pre: Set { p1, p2 }->forAll(GetAge() <= 30)"
    ]
    public void ForAll4()

```

```

{
}
OCL
[
  "context IteratorTests::ForAll5() : OclVoid"
  "pre: Set { p1, p2 }->forAll(px1; px2 | px1 <> px2
    implies px1.Age <> px2.Age)"
]
public void ForAll5()
{
}
OCL
[
  "context IteratorTests::ForAll6() : OclVoid"
  "pre: Set { p1, p2 }->forAll(px1 |
    Set { p1, p2 }->forAll(px2 | px1 <> px2
      implies px1.Age <> px2.Age))"
]
public void ForAll6()
{
}
OCL
[
  "context IteratorTests::IsUnique() : OclVoid"
  "pre: Set {1, 2, 3, 4, 5}->isUnique(t | t*2)"
]
public void IsUnique()
{
}
OCL
[
  "context IteratorTests::Any() : OclVoid"
  "pre: Set {1, 2, 3, 4, 5}->any(t | t > 2) > 2"
]
public void Any()
{
}
OCL
[
  "context IteratorTests::One() : OclVoid"
  "pre: Set {1, 2, 3, 4, 5}->one(t | t = 1)"
]
public void One()
{
}
OCL
[
  "context IteratorTests::Collect() : OclVoid"
  "pre: Set {1, 2, 3, 4, 5}->collect(t | t*2)->exists(p | p = 4)"
]
public void Collect()
{
}
OCL
[
  "context IteratorTests::Collect2() : OclVoid"
  "pre: Sequence {1, 2, 3, 4, 5}->collect(t | t*2)->
    exists(p | p = 4)"
]

```

```

public void Collect2()
{
}
OCL
[
    "context IteratorTests::CollectShort() : OclVoid"
    "pre: Set {p1, p2}.Age->exists(p | p = 10)"
]
public void CollectShort()
{
}
OCL
[
    "context IteratorTests::Select() : OclVoid"
    "pre: Set {1, 2, 3, 4}->select(t | t > 3).size() = 1"
]
public void Select()
{
}
OCL
[
    "context IteratorTests::Select2() : OclVoid"
    "pre: OrderedSet {1, 2, 3, 4}->select(t | t > 3).size() = 1"
]
public void Select2()
{
}
OCL
[
    "context IteratorTests::Select3() : OclVoid"
    "pre: Bag {1, 2, 3, 4, 4}->select(t | t > 3).size() = 2"
]
public void Select3()
{
}
OCL
[
    "context IteratorTests::Select4() : OclVoid"
    "pre: Sequence {1, 2, 3, 4, 4}->select(t | t > 3).size() = 2"
]
public void Select4()
{
}
OCL
[
    "context IteratorTests::Reject() : OclVoid"
    "pre: Set {1, 2, 3, 4}->reject(t | t > 3).size() = 3"
]
public void Reject()
{
}
OCL
[
    "context IteratorTests::Reject2() : OclVoid"
    "pre: OrderedSet {1, 2, 3, 4}->reject(t | t > 3).size() = 3"
]
public void Reject2()
{
}

```

```

OCL
[
  "context IteratorTests::Reject3() : OclVoid"
  "pre: Bag {1, 2, 3, 4, 4}->reject(t | t > 3).size() = 3"
]
public void Reject3()
{
}
OCL
[
  "context IteratorTests::Reject4() : OclVoid"
  "pre: Sequence {1, 2, 3, 4, 4}->reject(t | t > 3).size() = 3"
]
public void Reject4()
{
}
OCL
[
  "context IteratorTests::CollectNested() : OclVoid"
  "pre: Set {1, 2, 3, 4, 5}->collectNested(t | t*2)->exists(p | p = 4)"
]
public void CollectNested()
{
}
OCL
[
  "context IteratorTests::CollectNested2() : OclVoid"
  "pre: Sequence {1, 2, 3, 4, 5}->collectNested(t | t*2)->
    exists(p | p = 4)"
]
public void CollectNested2()
{
}
OCL
[
  "context IteratorTests::SortedBy() : OclVoid"
  "pre: Set {5, 4, 3, 2, 1}->sortedBy(t | t).at(1) = 1"
]
public void SortedBy()
{
}
OCL
[
  "context IteratorTests::SortedBy2() : OclVoid"
  "pre: Bag {5, 4, 5, 3, 2, 1}->sortedBy(t | t).at(1) = 1"
]
public void SortedBy2()
{
}
OCL
[
  "context IteratorTests::SortedBy3() : OclVoid"
  "pre: Sequence {5, 4, 5, 3, 2, 1}->sortedBy(t | t).at(1) = 1"
]
public void SortedBy3()
{
}
public void RunTests()
{
}

```

```

Console.WriteLine("----- ITERATOR TESTS -----");
Exists();
Console.WriteLine("I1 (Set {{1,2,3,4,5}}->exists(t | t > 4)): PASSED");
ForAll();
Console.WriteLine("I2 (Set {{1,2,3,4,5}}->forAll(t | t >= 1)): PASSED");
ForAll2();
Console.WriteLine("I3 (_pSet->forAll(p : Person | p.Age < 60)): PASSED");
ForAll3();
Console.WriteLine("I4 (Set {{p1, p2}}->forAll(Age < 30)): PASSED");
ForAll4();
Console.WriteLine("I5 (Set {{p1, p2}}->forAll(GetAge() < 30)): PASSED");
ForAll5();
Console.WriteLine("I6 (Set {{p1, p2}}->forAll(px1, px2 |
    px1 <> px2 implies px1.Age <> px2.Age)): PASSED");
ForAll6();
Console.WriteLine("I6 Set {{ p1, p2 }}->forAll(px1 |
    Set {{ p1, p2 }}->forAll(px2 | px1 <> px2
    implies px1.Age <> px2.Age)): PASSED");
IsUnique();
Console.WriteLine("I7 (Set {{1,2,3,4,5}}->
    isUnique(t | t *2)): PASSED");
Any();
Console.WriteLine("I8 (Set {{1, 2, 3, 4, 5}}->
    any(t | t > 2) > 2): PASSED");
One();
Console.WriteLine("I9 (Set {{1, 2, 3, 4, 5}}->
    one(t | t = 1)): PASSED");
Collect();
Console.WriteLine("I10 (Set {{1, 2, 3, 4, 5}}->
    collect(t | t*2)->exists(p | p = 4)): PASSED");
Collect2();
Console.WriteLine("I11 (Sequence {{1, 2, 3, 4, 5}}->
    collect(t | t*2)->exists(p | p = 4)): PASSED");
CollectShort();
Console.WriteLine("I12 (Set {{p1, p2}}.Age->
    exists(p | p = 10)): PASSED");
Select();
Console.WriteLine("I13 (Set {1, 2, 3, 4}->
    select(t | t > 3).size() = 1): PASSED");
Select2();
Console.WriteLine("I14 (OrderedSet {1, 2, 3, 4}->
    select(t | t > 3).size() = 1): PASSED");
Select3();
Console.WriteLine("I15 (Bag {1, 2, 3, 4, 4}->
    select(t | t > 3).size() = 2): PASSED");
Select4();
Console.WriteLine("I16 (Sequence {1, 2, 3, 4, 4}->
    select(t | t > 3).size() = 2): PASSED");
Reject();
Console.WriteLine("I17 (Set {1, 2, 3, 4}->
    reject(t | t > 3).size() = 3): PASSED");
Reject2();
Console.WriteLine("I18 (OrderedSet {1, 2, 3, 4}->
    reject(t | t > 3).size() = 3): PASSED");
Reject3();
Console.WriteLine("I19 (Bag {1, 2, 3, 4, 4}->
    reject(t | t > 3).size() = 3): PASSED");
Reject4();
Console.WriteLine("I20 (Sequence {1, 2, 3, 4, 4}->

```



```

        reject(t | t > 3).size() = 3): PASSED");
CollectNested();
Console.WriteLine("I21 (Set {{1, 2, 3, 4, 5}}->
    collectNested(t | t*2)->exists(p | p = 4)): PASSED");
CollectNested2();
Console.WriteLine("I22 (Sequence {{1, 2, 3, 4, 5}}->
    collectNested(t | t*2)->exists(p | p = 4)): PASSED");
SortedBy();
Console.WriteLine("I23 (Set {5, 4, 3, 2, 1}->
    sortedBy(t | t)->at(1) = 1): PASSED");
SortedBy2();
Console.WriteLine("I24 (Bag {5, 4, 5, 3, 2, 1}->
    sortedBy(t | t)->at(1) = 1): PASSED");
SortedBy3();
Console.WriteLine("I25 (Sequence {5, 4, 5, 3, 2, 1}->
    sortedBy(t | t)->at(1) = 1): PASSED");
Console.WriteLine("----- ITERATOR TESTS COMPLETE -----");
    }
}
// -----
// EOF

```

F.2 Navigation Expressions

```

// -----
// Nav.cs: Represents OCL testing for Navigation expressions
// -----
// Project: C# and OCL Compiler
// Module: Testing
// Author: Dave Arnold
// Version: 1.0
// -----
// -----
// Imports
// -----
using System;
using System.Collections;
// -----
// -----
// DaveArnold.OCLTesting Namespace
// -----
namespace DaveArnold.OCLTesting
{
    // A Class to test Navigation expressions
    class NavTests
    {
        public class Person
        {
            public string name = "Dave";
            public object data = new Object();
            public ArrayList ar = new ArrayList();
            public static int number = 5;
            public Person()
            {

```

```

        ar.Add(1);
        ar.Add(2);
        ar.Add(3);
    }
    OCL
    [
        "context NavTests::Person::NumberOfNames() : Integer"
        "pre: self.name->size() = 1"
        "post: result = self.name->size()"
    ]
    public int NumberOfNames()
    {
        return 1;
    }
    OCL
    [
        "context NavTests::Person::GetNumber() : Integer"
        "post: result = Person::number"
    ]
    public int GetNumber()
    {
        return Person.number;
    }
    OCL
    [
        "context NavTests::Person::GetMaxNumber() : Integer"
        "post: result = Int32::MaxValue"
    ]
    public int GetMaxNumber()
    {
        return Int32.MaxValue;
    }
    OCL
    [
        "context NavTests::Person::GetData() : System::Object"
        "pre: not self.data->isEmpty()"
    ]
    public object GetData()
    {
        return data;
    }
    OCL
    [
        "context NavTests::Person::ArCount() : Integer"
        "post: result = ar->size()"
    ]
    public int ArCount()
    {
        return ar.Count;
    }
}
public class PersonData
{
    public Person px = new Person();
}
    OCL
    [
        "context NavTests::GetName(px : PersonData) : String"
        "post: result = px.px.name"
    ]

```

```

    }
    public string GetName(PersonData px)
    {
        return px.px.name;
    }
    OCL
    [
        "context NavTests::GetLength(px : PersonData) : Integer"
        "post: result = px.px.name.Length"
    ]
    public int GetLength(PersonData px)
    {
        return px.px.name.Length;
    }
    OCL
    [
        "context NavTests::GetLengthX2(px : PersonData) : Integer"
        "post: result = px.px.name.Length * 2"
    ]
    public int GetLengthX2(PersonData px)
    {
        return px.px.name.Length * 2;
    }
    public void RunTests()
    {
        Console.WriteLine("----- NAV TESTS -----");
        Console.WriteLine("N1 (GetName()): {0}", GetName(new PersonData()));
        Console.WriteLine("N2 (GetLength()): {0}",
            GetLength(new PersonData()));
        Console.WriteLine("N3 (GetLengthX2()): {0}",
            GetLengthX2(new PersonData()));
        Person p = new Person();
        Console.WriteLine("N4 (p.NumberOfNames()): {0}", p.NumberOfNames());
        Console.WriteLine("N5 (p.GetData()): {0}", p.GetData());
        Console.WriteLine("N6 (p.ArCount()): {0}", p.ArCount());
        Console.WriteLine("N7 (p.GetNumber()): {0}", p.GetNumber());
        Console.WriteLine("N8 (p.GetMaxNumber()): {0}", p.GetMaxNumber());
        Console.WriteLine("----- NAV TESTS COMPLETE -----");
    }
}
// -----
// EOF

```

F.3 Casting and State Expressions

```

// -----
// Cast.cs: Represents OCL testing for casting expressions
// -----
// Project: C# and OCL Compiler
// Module: Testing
// Author: Dave Arnold

```

```

// Version: 1.0
// -----
// -----
// Imports
// -----
using System;
using System.Collections;
// -----
// -----
// DaveArnold.OCLTesting Namespace
// -----
namespace DaveArnold.OCLTesting
{
    // A Class to test the casting expressions
    class CastTests
    {
        private object p = null;
        OCL
        [
            "context CastTests::Length(o : System::Object) : Integer"
            "pre: not o.ocIsUndefined()"
            "pre: o.ocIsInState(o)"
            "pre: o.ocIsTypeOf(Object)"
            "pre: o.ocIsKindOf(String)"
            "post: result = o.ocAsType(String).Length"
            "post: p.ocIsNew()"
        ]
        public int Length(object o)
        {
            p = new object();
            return (o as String).Length;
        }
        public void RunTests()
        {
            Console.WriteLine("----- CAST TESTS -----");
            Console.WriteLine("C1 (Length((object)\\"Hello\\")): {0}",
                Length((object)"Hello"));
            Console.WriteLine("----- CAST TESTS COMPLETE -----");
        }
    }
}
// -----
// EOF

```