I would like to thank Dr. Alessandra Russo for supervising my project. Her immense enthusiasm and guidance throughout the whole period has enabled me to produce a project that should be of use to the software community.

I would also like to thank Dr. Krysia Broda for her general advice and criticisms which have helped me to refine my project ideas, inevitably leading to a more complete project.

Finally, I’d like to thank Jos Warmer and Anneke Kleppe for their continual efforts in developing OCL, and promoting it to the wider community.
Abstract

“The Unified Modelling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modelling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modelling of large and complex systems.”

Rational Software Corporation

“The Object Constraint Language (OCL) is an expression language that enables one to describe constraints on object-oriented models and other object modelling artefacts.”

Anneke Kleppe

This project involves the design and implementation of a tool which allows the user to create UML class diagrams, to add OCL constraints and to facilitate a series of consistency checks across both of these inputs.
## List Of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>‘Person’ class diagram</td>
</tr>
<tr>
<td>2.1</td>
<td>Basic Structure of an OCL file</td>
</tr>
<tr>
<td>2.2</td>
<td>‘Account’ class diagram</td>
</tr>
<tr>
<td>2.3</td>
<td>Typical XML representation of the Account class</td>
</tr>
<tr>
<td>2.4</td>
<td>Screenshot of UML class diagrams created using ‘Rational Rose’</td>
</tr>
<tr>
<td>3.1</td>
<td>The main stages and data flows involved in our system</td>
</tr>
<tr>
<td>4.1</td>
<td>The UML construct data structures</td>
</tr>
<tr>
<td>4.2</td>
<td>The Project Data Structure</td>
</tr>
<tr>
<td>4.3</td>
<td>Main layout of Rational Rose</td>
</tr>
<tr>
<td>4.4</td>
<td>Main layout of Borland JBuilder</td>
</tr>
<tr>
<td>4.5</td>
<td>Layout sketch for user interface</td>
</tr>
<tr>
<td>4.6</td>
<td>Project Structure using a Directory Tree</td>
</tr>
<tr>
<td>4.7</td>
<td>Main library classes involved in the directory tree</td>
</tr>
<tr>
<td>4.8</td>
<td>Properties table for the ‘Class’ construct</td>
</tr>
<tr>
<td>4.9</td>
<td>Template Class Diagram</td>
</tr>
<tr>
<td>4.10</td>
<td>Template Interface Diagram</td>
</tr>
<tr>
<td>4.11</td>
<td>Template Association Diagram</td>
</tr>
<tr>
<td>4.12</td>
<td>A typical rendered Class produced using our rendering algorithm</td>
</tr>
<tr>
<td>4.13</td>
<td>Screen shot of the Development Environment</td>
</tr>
<tr>
<td>6.1</td>
<td>Example list of characters</td>
</tr>
<tr>
<td>6.2</td>
<td>Example set of tokens</td>
</tr>
<tr>
<td>6.3</td>
<td>Example AST</td>
</tr>
<tr>
<td>6.4</td>
<td>Interaction of parser components</td>
</tr>
<tr>
<td>6.5</td>
<td>The Node Hierarchy</td>
</tr>
<tr>
<td>6.6</td>
<td>Error Message Format</td>
</tr>
<tr>
<td>7.1</td>
<td>Example AST representation of 5 + 2.7 * 3</td>
</tr>
<tr>
<td>7.2</td>
<td>Example type tree representation of 5 + 2.7 * 3</td>
</tr>
<tr>
<td>7.3</td>
<td>The Type Hierarchy</td>
</tr>
<tr>
<td>7.4</td>
<td>The ‘Environment’ Interface</td>
</tr>
<tr>
<td>7.5</td>
<td>The ‘UMLModel’ Interface</td>
</tr>
<tr>
<td>7.6</td>
<td>Error Message Format</td>
</tr>
<tr>
<td>7.7</td>
<td>‘Person’ Class Diagram</td>
</tr>
<tr>
<td>7.8</td>
<td>‘Company’ Class Diagram</td>
</tr>
<tr>
<td>8.1</td>
<td>‘Person’ Class Diagram</td>
</tr>
<tr>
<td>8.2</td>
<td>‘Person’ Class Diagram Revisited</td>
</tr>
<tr>
<td>8.3</td>
<td>‘ClassA’ Class Diagram</td>
</tr>
<tr>
<td>9.1</td>
<td>Official OCL 1.4 Specification example Class Diagram</td>
</tr>
<tr>
<td>9.2</td>
<td>The Classes menu bar</td>
</tr>
<tr>
<td>9.3</td>
<td>The Project Directory tree after the class creation</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>9.4</td>
<td>The class properties table after the name change</td>
</tr>
<tr>
<td>9.5</td>
<td>The Project Directory tree after the attribute creation</td>
</tr>
<tr>
<td>9.6</td>
<td>The attribute properties table after the name and type change</td>
</tr>
<tr>
<td>9.7</td>
<td>The Project Directory tree after the method creation</td>
</tr>
<tr>
<td>9.8</td>
<td>The method properties table after the property changes</td>
</tr>
<tr>
<td>9.9</td>
<td>Microsoft Visual Studio Properties table – properties are listed vertically</td>
</tr>
<tr>
<td>9.10</td>
<td>The Associations menu bar</td>
</tr>
<tr>
<td>9.11</td>
<td>The association properties table after the property changes</td>
</tr>
<tr>
<td>9.12</td>
<td>The entire project tree</td>
</tr>
<tr>
<td>9.13</td>
<td>The Tools menu bar</td>
</tr>
<tr>
<td>9.14</td>
<td>UML Validity Output for our example</td>
</tr>
<tr>
<td>9.15</td>
<td>AST representation of our constraint</td>
</tr>
<tr>
<td>9.16</td>
<td>Performance Graph</td>
</tr>
</tbody>
</table>
# Contents

Acknowledgments i  
Abstract ii  
List of Figures iii  
Contents v  

## 1. Introduction  
1.1 Overview 1  
1.2 My Contribution 3  
1.3 Report Structure 3  

## 2. Background  
2.1 Basics of OCL constraints 4  
2.2 UML Constructs 5  
2.3 Existing Software 6  
2.5 Parser Technologies 7  
2.5 Type Checking Methodologies 8  
2.6 Taking User Input 10  
2.7 Summary 12  

## 3. System Overview  
3.1 What Consistency Checks Will We Cover 13  
3.2 Design Decisions 13  
3.3 Main Components 14  

## 4. Development Environment  
4.1 Overview 16  
4.2 Class Diagram Input 16  
4.3 Main Data Structures 17  
4.4 Taking User Input 19  
4.5 Required Visual Components 20  
4.6 Analysis of Visual Components 22  
4.7 GUI Layout Design 22  
4.8 GUI Implementation 23  
4.9 Project / Directory Tree 23  
4.10 Properties Table 25  
4.11 Main Display Area 25  
4.12 Information / Message Tabs 27  
4.13 File Persistence 28  
4.14 Summary 29  

## 5. Class Diagram Validation  
5.1 Overview 30  
5.2 Class Validation 30  
5.3 Class Attribute Validation 31  
5.4 Class Method Validation 31  
5.5 Interface Validation 32
1. Introduction

1.1 Overview

In recent years UML has become a very popular modelling language for software systems. It consists of a number of modelling techniques, one of which is the use of class diagrams, which are used to document the static structure of a system; that is, what classes there are and how they are related.

OCL differs in that it is instead a formal language that allows the specification of constraints over UML class diagrams.

Various tools are currently available for supporting the development of UML specifications, however, little has been done in integrating UML with OCL constraints.

This project involves the design and implementation of a tool which allows the user to create UML class diagrams, to add OCL constraints and to facilitate a series of consistency checks across both of these inputs.

What exactly are class diagrams?

The UML notation is rich and full bodied. It is comprised of two major subdivisions. There is a notation for modelling the dynamic elements of a design such as objects, messages, and finite state machines. There is also a notation for modelling the static elements of a design such as classes, attributes, and relationships. Typical static models are presented in UML as diagrams called Class Diagrams.

In an object orientated application classes have attributes, operations and relationships with other classes. The fundamental element of a class diagram is an icon that represents a class. A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes, and the bottom compartment contains a list of operations. An example class diagram can be seen in figure 1.1. This models a class called ‘Person’ which has a ‘name’ attribute and a method for accessing this attribute called ‘getName’.

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : String</td>
</tr>
<tr>
<td>getName() : String</td>
</tr>
</tbody>
</table>

figure 1.1 ‘Person’ class diagram

What exactly is OCL?

OCL is an expression language that enables one to describe constraints on object-oriented models and other object modelling artifacts. A constraint is a restriction on one or more values of an object-oriented model or system.
An example constraint on the class diagram in *figure 1.1* would be to say that the length of a Person’s name must always be less than twenty characters. This can be expressed in OCL by using the following constraint:

```
context Person inv:
  self.name.length() < 20
```

**Why use OCL?**

Class diagrams typically specify the main elements of the entities which they are modelling but fail to provide all the relevant aspects of the specification.

Designers usually need to add constraints to the model to cater for this weakness of class diagrams, these constraints are often described in natural language.

However, constraints written using natural language inevitably lead to ambiguities, and as a consequence, a series of formal languages have been developed. These languages are mathematically based therefore, for the average systems modeller, they are difficult to use. OCL was developed to fill this gap. It provides an easy to use syntax but still contains many features offered by rival formal languages.

**What do we mean by Consistency Checks?**

In other words, are the constraints consistent with the UML class diagram? This amounts to two specific issues:

i. Do all class diagram entities referenced in the constraints actually exist in the class diagram.

ii. Type conformance.

If we consider the class diagram used in *figure 1.1* and the following constraint:

```
context Person inv:
  self.name = 10
```

Then obviously we have a type incompatibility as we can’t compare an Integer with a String. The other element of consistency checking is determining whether all class diagram entities referenced in the constraints actually exist in the class diagram. So in our example, we would need to check whether the type ‘Person’ and the attribute ’name’ had actually been defined in the class diagram. However, checking for type conformance would involve resolving all references to the class diagram, and so this process would detect any invalid references. Therefore, type checking would naturally solve both of these issues.

The example demonstrates the challenges we may encounter in the simplest of cases. When multiple classes are used, together with associations, and constraints are specified with the full range of OCL syntax, then the task of consistency checking becomes rather more complex.
1.2 My contribution

This project was established with the aim of developing an environment whereby users can enter UML class diagrams, add OCL constraints and conduct a series of consistency checks across both of these specifications.

Overall a tool was developed which allows the user to:

i. Create UML class diagrams

ii. Validate class diagrams making sure that they are consistent with the UML 1.4 (latest) specification.

iii. Add OCL constraints

iv. Validate OCL constraints making sure that they conform to the OCL 1.4 (latest) grammar.

v. Check for type conformance between both inputs – Approximately 95% grammar coverage.

Also, numerous semantic issues involving consistency checks have been highlighted.

1.3 Report Structure

The rest of the report covers the research, design, implementation and evaluation of a suite of tools which aid the integration of UML class diagrams and OCL constraints. This report can also be found online at: www.doc.ic.ac.uk/~dms99/project/report.doc

Chapter 2 covers the background theory behind the project and highlights related research and existing techniques.

Chapter 3 gives an overview of the development tools which this project aims to create and briefly discusses the main design decisions.

Chapter 4 profiles the design and implementation of the ‘Development Environment’.

Chapter 5 profiles the design and implementation of the ‘Class Diagram Validator’.

Chapter 6 profiles the design and implementation of the ‘OCL Parser’.

Chapter 7 profiles the design and implementation of the ‘Type Checker’.

Chapter 8 discusses various semantic issues which affect the meaning of typical constraints.

Chapter 9 evaluates how well the system meets the requirements and goals of the project when tested with a case study.

Chapter 10 concludes the report with a short discussion on the achievement of this project and a brief look at possible future improvements and enhancements.
2. Background

2.1 Basics of OCL constraints

This section aims to cover the essentials of OCL constraints without going into too much detail. We discuss the simple format of an OCL file and then go onto look at the four main types of statement. We finish off with a look at the range of OCL types. Heavy use is made of the OCL 1.4 specification [OCL1.4].

**Basic File Structure**

All OCL files begin with a simple ‘package’ statement which groups together the succeeding constraints. A package is then made up of a number of ‘context’ statements which contain a series of constraints – obviously these constraints are written in terms of the context specified by the ‘context’ statement. *Figure 2.1* shows this structure.

```
package ThePackage

context A
  // A series of constraints
context B
  // A series of constraints
...
endpackage
```

*figure 2.1 Basic Structure of an OCL file*

**Types of constraints**

OCL offers three types of constraints on class diagrams with a fourth offering the user the ability to extend the class diagram:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-condition</td>
<td>This is specified using a ‘pre’ statement and allows the user to state the conditions which must hold at the beginning of a method execution</td>
</tr>
<tr>
<td>Post-condition</td>
<td>This is specified using a ‘post’ statement and allows the user to state the conditions which must hold at the end of a method execution</td>
</tr>
<tr>
<td>Invariant</td>
<td>This is specified using an ‘inv’ statement and allows the user to state the conditions which must be true at all times</td>
</tr>
<tr>
<td>Definition</td>
<td>This is specified using the ‘def’ statement and allows the user to specify additional attributes / operations on the UML class diagram</td>
</tr>
</tbody>
</table>
The OCL syntax allows expressions to be created, all of which must evaluate to a specific type which exists either in the UML model or within OCL. Typical expressions are formed by the navigation of the class diagram.

All available types can be grouped into four distinct categories:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UML Model Types</strong></td>
<td>These are all types which have been defined within the UML class diagram</td>
</tr>
<tr>
<td><strong>OCL Basic Types</strong></td>
<td>These consist of the four types: (i) Integer (ii) Real (iii) String (iv) Boolean</td>
</tr>
<tr>
<td><strong>OCL Collection Types</strong></td>
<td>These consist of the four types: (i) Collection (ii) Set (iii) Sequence (iv) Bag</td>
</tr>
<tr>
<td><strong>OCL Meta-level Types</strong></td>
<td>The basic OCL types are supplemented by three other types:</td>
</tr>
<tr>
<td></td>
<td>i.  <em>OclExpression</em> – Which represents all expressions used within OCL</td>
</tr>
<tr>
<td></td>
<td>ii. <em>OclType</em> – The parent type of all types in the UML class diagram</td>
</tr>
<tr>
<td></td>
<td>iii. <em>OclAny</em> – The super type of all UML types and the OCL Basic types</td>
</tr>
<tr>
<td></td>
<td>These allow the modeller to gain access to the meta-level of the OCL model</td>
</tr>
</tbody>
</table>

### 2.2 UML Constructs

We will use the term *Construct* to refer to a particular building block in a class diagram. The UML 1.4 specification [UML1.4] hosts a plethora of constructs, some of which OCL does not concern itself with. In order to reduce the complexity of the class diagram it would be worth investigating exactly what constructs OCL does cater for.

The following table lists details of the UML constructs which feature in the OCL specification [OCL1.4] and hence can be used within OCL expressions:

<table>
<thead>
<tr>
<th>UML Construct</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td>This is the most essential construct. It comes with attributes and methods</td>
</tr>
<tr>
<td><strong>Association</strong></td>
<td>OCL caters for three variations of associations:</td>
</tr>
<tr>
<td></td>
<td>i.  Normal associations between 2 classes</td>
</tr>
<tr>
<td></td>
<td>ii. Association classes which are attached to associations</td>
</tr>
<tr>
<td></td>
<td>iii. Qualified associations</td>
</tr>
<tr>
<td><strong>Enumeration</strong></td>
<td>A data type whose range is a list of predefined values</td>
</tr>
</tbody>
</table>
2.3 Existing Software

This section profiles several existing systems which already tackle some of the issues in this project.

**Rational Rose – Rational Software Corporation [RROSE]**

This is a UML development tool which offers the user the ability to create UML class diagrams. It is fully compliant with the UML 1.4 specification [UML1.4]. It offers no core support for OCL.

**Microsoft Visio – Microsoft Corporation [MSVIS]**

This is another software design tool which allows the user to create any kind of diagram including class diagrams. This too offers no support for OCL.

**Modular OCL Compiler – Frank Finger [FF2000]**

Frank Finger at Dresden University has developed a tool which does full type checking of OCL constraints and includes code generation from OCL to Java. Finger’s tool can be difficult to use by itself as it doesn’t offer any class diagram creation capabilities, however within just the last few months it has apparently been integrated with a UML development tool called ‘Argo’ [ARGO].

Finger’s tool is only compatible with the OCL 1.3 specification and was developed as part of a Diploma thesis which took several years to complete.

**Runtime evaluation of OCL – David Akehurst [AKE]**

David Akehurst from Canterbury University has developed a Java library that enables runtime evaluation of OCL [AKE].

This approach to providing OCL support within the Java programming language, enables runtime creation and evaluation of OCL expressions. It uses a combination of reflection and runtime class creation to construct objects that represent OCL expressions. These expression objects can be treated like any other Java object, and can be evaluated in a specified context.

Like the previous tool, no support is available to aid the development of UML class diagrams. Currently, only a partial implementation has been provided and this is also inconsistent with the latest OCL specification.

**Summary**

It is evident that there exist a number of systems which already tackle some of the problems faced in this project. However, apart from the recent integration of the ‘Argo’ tool with Frank Finger’s OCL compiler (which occurred near the end of this project), there does not appear to be a general tool which offers an overall ‘UML and OCL’ driven design environment. This project aims to fill that gap.
2.4 Parser Technologies

At some stage we will need to validate the OCL input. The first stage will involve checking the syntax of the constraints i.e. whether they conform to the grammar. The most obvious choice of technique is to construct a parser which will scan the input and check for grammar conformance.

Writing a parser from scratch would be some what of a lengthy process, fortunately there exist a number of libraries which provide the basic framework for parser construction. The tool will most likely be implemented in Java, so in this section we review two of the main open source parser libraries for use with the Java programming language.

SableCC

This is an object-oriented framework that uses techniques to automatically build a strictly typed abstract syntax tree that matches the grammar of the compiled language. It also generates tree-walker classes using an extended version of the visitor design pattern which enables the implementation of actions on the nodes of the abstract syntax tree using inheritance [SABCC].

Strengths

i. Generates AST’s

ii. Generates Tree walker classes

iii. Full Unicode Support – Very useful if the parser is needed to support a large range of characters i.e. different languages.

iv. LALR based parsers - LALR, or "lookahead LR" parsers produce relatively small parsing tables, thus it has very significant advantages in size over its LR counterpart.

Weaknesses

i. Little support – Compared to other rival parsers, SableCC has less resources available to developers.

JavaCC

This is another object orientated framework for parser generation. Like SableCC, it produces AST’s and tree walker classes [JAVACC]. This was originally developed by Sun Microsystems – the authors of Java.

Strengths

i. Same as SableCC
ii. **Large User community** – This is considered to be the most popular parser generator in existence. There are a whole host of resources available which should make development easier.

iii. **User friendly** – Comes with an easy-to-use GUI which simplifies the parser creation process.

**Weaknesses**

i. **Large Files** – Due to the complexity of the parser, JavaCC in particular produces a lot of code which can make it difficult to understand the internal details of the parser.

### 2.5 Type Checking Methodologies

The main element of our consistency checks will involve static type checking. This will make sure that all types used in the OCL constraints match and that they do exist either within the OCL type hierarchy or within the UML class diagram.

A successful parse of the OCL constraints will result in an AST representation of them. Going back to the second year ‘Compilers’ course [RBCOM], type checking an AST basically involves evaluating the type of all nodes in the tree. This section discusses various ways of evaluating the type of all nodes in an AST.

**Visitor Pattern**

This enables us to represent an operation to be performed on the elements of an object structure. It permits a new operation to be defined without changing the classes of the elements on which it operates [GOF].

In the case of type checking, we will have a class representing each type of node. Each of these nodes will have an `accept` method which will be defined as follows:

```java
void accept(Visitor v) {
    v.visit(this);
}
```

We would have a `Visitor` interface which lists the methods which a ‘Visitor’ would need to implement:

```java
interface Visitor {
    void visit(ASTnode1 n);
    void visit(ASTnode2 n);
    ...
}
```

Finally we would need to define the visitor class which implements the `Visitor` interface. This class will contain the actual type checking code for each particular node:

```java
Class StaticTypeChecker implements Visitor {
    void visit(ASTnode1 n) {
        // type checking code for ASTnode1
```
void visit(ASTnode2 n) {
    // type checking code for ASTnode2
    ...
}

Strengths

i. **Gathers related operations** – All type checking code is located in the same place.

ii. **Flexible** – The framework makes adding new operations easy. For instance we may want to add another level of type checking which would only require us to define a new Visitor class (as well as the relevant type checking code).

Weaknesses

i. **Difficult to add new Node classes** – When adding new Node classes we would have to extend the framework to handle these which is time consuming.

ii. **Can break encapsulation** – The Visitor classes must have access to enough visited element state to perform their function.

iii. **Large type checking file** – Despite the grouping of type checking code being an advantage, if the type checking becomes complex and lengthy then placing this all in one file may make the program difficult to understand.

Multiple ‘check’ methods

This method is based on a similar methodology demonstrated in the second year ‘Compiler’s’ course [RBCOM] where a ‘check’ method is implemented separately for each node.

Assuming again that we have a different class for each type of node in the AST, we define a type checking class for each node. Each of these classes would yield a custom ‘check’ method which handles the type checking of that specific node.

For example, if we have a node called ASTnode1 then we could define another class called CheckASTnode1 which will contain the appropriate check method:

```java
class CheckASTnode1 {
    public static Type check(ASTnode1 n, ...) {
        // Type checking code
    }
}
```

Type checking the ASTnode1 would simply involve calling check(ASTnode1 n, ...) with the appropriate arguments.

**Strengths**


i. **Separates out code** – Type checking code is separated out into manageable classes which stay at a relatively small size. All type checking classes will also exist in one package.

ii. **Custom ‘check’ signatures** – Each node can have a custom check signature (this is difficult to achieve in the visitor pattern).

**Weaknesses**

i. **Less elegant** – Not very object orientated as compared to the Visitor Pattern.

### 2.6 Taking User Input

Essentially any solution will need to be able to take the UML and OCL input from the user. This section looks at the different ways in which this can be achieved, relating these methods to existing software systems.

**Simple text input/file**

In conventional programming languages, the means by which user input is passed to the system (or compiler) is through a simple text file. In creating web documents, the document information is specified using HTML. The HTML is parsed by a web browser where it is converted into an internal representation and then processed.

These same principles could be duplicated in our project. The input could be specified using an XML type syntax and then parsed using one of a number of existing XML parsers.

Consider the ‘Account’ class in **figure 2.2**

```
<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance : Real</td>
</tr>
<tr>
<td>add(r : Real) : Void</td>
</tr>
<tr>
<td>remove(r : Real) : Void</td>
</tr>
</tbody>
</table>
```

**figure 2.2 ‘Account’ class diagram**

The information could be grouped into three categories:

i. **Class info** – i.e. it’s name, is it abstract, which other classes does it extend etc.

ii. **Attribute info** – Details of the attributes that it contains.

iii. **Method info** – Details of the methods that it contains.

**Figure 2.3** shows a typical XML representation of the Account class. Such a file could then be fed into a typical XML parser such as ‘Xerces’ [XER] and then relatively easily converted into a system representation.
Strengths

i. **Relatively straightforward** to implement – We can use existing parser libraries to help us.

ii. **Flexible** – If more information needs to be specified about the class then we can create more tags. Most XML parsers are designed to cope with such changes.

iii. Fairly **System independent** – Other UML based systems could share this format.

Weaknesses

i. **Non-user friendly** – A big disadvantage of this approach is that users would need an understanding of XML and a specific knowledge of the file format being used. Also, whilst constructing the model the user has no visual feedback on the current state.

**GUI Approach**
This type of approach is typical of existing UML based design systems, such as ‘Rational Rose’ [RROSE] and ‘Microsoft Visio’ [MSVIS]. Users create class diagrams on the fly by using the mouse to drag and drop shapes and then by entering specific information. These types of systems are highly interactive and user friendly.

*Figure 2.4* shows a typical class diagram created using ‘Rational Rose’. The actual classes are created by selecting boxes from a panel of shapes and dragging these onto the drawing canvas. The class information is then entered into these boxes. Associations are simply added by drawing lines between the classes and then labelling these.

A typical GUI implementation will not need to be as advanced as existing UML development software but should at least allow the user to directly input class diagram information on screen.

**Strengths**

i. Very **User Friendly** – Easy to develop and edit diagrams, users don’t need much knowledge of the software in order to create them

ii. **Visualisation of model** – Users have a consistent graphical view of what they’re creating

**Weaknesses**

i. **Difficult to implement** – Due to the high user interactivity a lot of research needs to go into the ‘human-computer interfacing’ aspect. Also, the general concept of dragging and dropping shapes and dynamically creating diagrams is demanding in terms of design and implementation.

### 2.7 Summary

In this section we have covered several of the key problems which relate to our project. For each of these problems we have outlined several solutions, the next chapter details our choices of solutions.
3. System Overview

3.1 What Consistency Checks Will We Cover

So far the phrase ‘consistency checks’ has been used in a very general context. The consistency checks that this tool will cover amount to the process of type checking i.e. making sure that all types used in the OCL constraints conform. From now on we will refer to our consistency checking as ‘type checking’.

3.2 Design Decisions

This section describes the design decisions made after the background research was carried out.

Choice of Language

There are several languages to choose from for the implementation. The tool would require significant data processing and extensive user interaction.

C/C++ could easily cope with both of these features but would be demanding for someone with little experience with these technologies.

Visual Basic offers fast GUI building capabilities but due to the large data processing requirement would inevitably have a performance lag.

Java could also deal with both requirements but due to its interpreted nature would run slower than compiled C/C++. However, Java offers a whole host of easy to use GUI libraries and is considered to be simpler than C++, so this will be the language that we will use for the implementation.

Choice of Parser Library

This is a difficult choice as the two main alternatives listed in section 2.4 have similar strengths and weaknesses. However, JavaCC will be used to aid the parser construction process. This is because the basic parser which will be constructed for us will need to be customised to our particular needs, and JavaCC offers a better combination of resources and examples.

Choice of Type Checking Methodology

Despite the elegance of the Visitor pattern, the second option of using ‘multiple check methods’ will be used. This option allows all type checking code related to each node to reside in a separate, independent class.

Choice of User Input Method

The simple ‘text input / file’ option offers a quick and easy implementation but has the downside of being very non-user friendly and would be difficult for new users to grasp. In
contrast, the GUI approach is very user friendly but would require a lengthy, complex implementation.

Due to the existing functionality of UML design tools which are currently on the market, it would be expected that this tool should have some sort of GUI that would aid the development of class diagrams, as a text file input really would be too restrictive and impractical for most users. Upon this thought, a variant of the GUI approach will be used whereby users can graphically create class diagrams but the ‘creation’ process will be fairly basic, so that the implementation will not be too lengthy.

3.3 Main Components

In order to check the consistency of OCL constraints in relation to a UML class diagram, there will be a set number of stages of computation. Figure 3.1 helps us visualise these steps. The first stage involves taking the input from the user and converting this into an internal representation, this will be one of the main tasks of the Development Environment. Section 2.6 discussed various ways of achieving this.

Once we have our UML class diagram and OCL constraints in the system we then check the validity of each of these independently i.e. we check for conformance with each of the respective specifications. This stage will either generate an error indicating where each of the inputs has failed to comply with it’s specification, or, will pass the whole input to the next stage. The validation of the OCL constraints will need the use of a parser which will check for grammar conformance. The validation of the UML class diagram will amount to a series of syntactic and semantic checks.

The third and final stage involves the type checking of both inputs in relation to each other. The output of this stage will be considered as the output of the system. We will either receive a message indicating a successful execution or an error message detailing any type errors between the two inputs.
figure 3.1. The main stages and data flows involved in our system
4. Development Environment

4.1 Overview

Creating the development environment will be the first problem we will tackle. The development environment will have two main purposes. The first is to take the relevant input from the user and to convert this into an internal representation, which can be processed at a later stage. The second purpose is to provide functionality similar to existing Integrated Development Environment’s (IDE’s) such as ‘Microsoft Visual Studio’ [MSVS] whereby users can create and edit projects and can use certain tools on the data they enter. In other words, this will be the central system which brings together all tools which will be developed throughout this project. The development environment will be what the user directly interacts with, therefore a strong emphasis will be placed on the user interface.

In this chapter we initially cover the various types of UML constructs that we let the user have access to, and the required data structures needed to store instances of these constructs. We then describe specifically how the user is expected to enter their model information and then we review the necessary components / features that a typical development environment should offer. We discuss possible interface layouts, carefully profiling these against existing systems. A basic implementation of the layout using Java Swing technology will follow.

The design and implementation of each of the components will fill most of the chapter, we then round things of with a look at how we tackle the problem of file persistence.

4.2 Class Diagram Input

Due to the sheer size of the UML specification and the large number of constructs that UML offers we will limit our tool to only offer a predefined number of constructs. These constructs will come in two categories, those which OCL is aware of and those which are considered essential to UML.

The constructs which OCL is aware of were documented in section 2.2, once again they are:

i. Class
ii. Association
iii. Enumeration

Another construct which is considered essential in UML is the ‘Interface’. Despite OCL’s inability to use this construct we include this in our tool for completeness.

As enumerations have little importance in OCL these will not be catered for. This leads us to the following list of UML constructs which our tool will handle:

i. Class
ii. Interface
iii. Association
As well as defining new instances of these constructs, the user will have the ability to specify particular properties for them. For example, for the ‘Class’ construct the user can state which other classes it extends, whether it is abstract or not etc. The UML 1.4 specification [UML1.4] was used to help determine which properties / elements of these constructs were required. These are listed in the following table:

<table>
<thead>
<tr>
<th>Construct</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Abstract</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Extends</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Association Class</strong></td>
</tr>
<tr>
<td>Class Attribute</td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Access</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Class Method</td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Access</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Abstract</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Static</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return type</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Parameter types</strong></td>
</tr>
<tr>
<td>Interface</td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>Interface Method</td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return type</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Parameter types</strong></td>
</tr>
<tr>
<td>Association</td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Association Class</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Left Entity</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Left Label</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Left Multiplicity</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Right Entity</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Right Label</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Right Multiplicity</strong></td>
</tr>
</tbody>
</table>

4.3 Main Data Structures

In a typical project we will need to keep track of the various UML constructs which we have created. For each of these constructs a class will be required which will hold the relevant data for a particular instance of that construct. These data classes will be a simple translation
of the information presented in section 4.2. Each data class will contain a list of attributes corresponding to the properties of a construct, and a series of get and set methods which are used to access and modify these properties. These classes can be seen in figure 4.1.

<table>
<thead>
<tr>
<th>ClassEssential</th>
<th>ClassAttribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : String</td>
<td>name : String</td>
</tr>
<tr>
<td>abstract : Boolean</td>
<td>type : String</td>
</tr>
<tr>
<td>extends : String</td>
<td>access : String</td>
</tr>
<tr>
<td>associationClass : Boolean</td>
<td>// corresponding get and set methods</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GeneralMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : String</td>
</tr>
<tr>
<td>returnTypes : String[]</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ClassMethod</th>
<th>InterfaceMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract : Boolean</td>
<td></td>
</tr>
<tr>
<td>static : Boolean</td>
<td></td>
</tr>
<tr>
<td>access : String</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>InterfaceEssential</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : String</td>
<td>name : String</td>
</tr>
<tr>
<td></td>
<td>leftEntity : String</td>
</tr>
<tr>
<td></td>
<td>leftLabel : String</td>
</tr>
<tr>
<td></td>
<td>leftMultiplicity : String</td>
</tr>
<tr>
<td></td>
<td>rightEntity : String</td>
</tr>
<tr>
<td></td>
<td>rightLabel : String</td>
</tr>
<tr>
<td></td>
<td>rightMultiplicity : String</td>
</tr>
<tr>
<td></td>
<td>// corresponding get and set methods</td>
</tr>
</tbody>
</table>

*figure 4.1 The UML construct data structures*
We will also need a data structure which will store all the information relating to a project. A typical project will be composed of several items, which are:

i. **Name** – This identifies the project.
ii. **UML constructs** – A list of UML constructs which have been created.
iii. **OCL Constraints** – The various OCL constraints which have been entered.
iv. **Documentation Notes** – The notes which have been added by the user.

A simple data structure capable of holding this information can be seen in figure 4.2

<table>
<thead>
<tr>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : String</td>
</tr>
<tr>
<td>umlConstructs : Tree</td>
</tr>
<tr>
<td>oclConstraints : String</td>
</tr>
<tr>
<td>docNotes : String</td>
</tr>
<tr>
<td>// corresponding get and set methods</td>
</tr>
</tbody>
</table>

*figure 4.2 The Project Data Structure*

### 4.4 Taking User Input

Before we go into the details of creating the user interface, we discuss how the user will interact with a general user interface in order to input their model information and constraints.

**Class Diagram creation**

A UML development tool like ‘Rational Rose’ [RROSE] offers a very user friendly class diagram creation process. This subsection extracts the general nature of this process with the hope of reusing this methodology in our system.

A class diagram is made up of many constructs, the process of creating and customising new instances of these constructs can broken down into two simple steps:

i. The user selects the required construct. This action should create a new instance of that construct.

ii. The user then customises the construct by modifying it’s properties.

**Constraint specification**

Essentially, the specification of OCL constraints is analogous to writing code. Taking these constraints simply amounts to giving the user a text area to write them in.
4.5 Required Visual Components

We now look at the basic requirements of our interface and produce a list of visual components which will be needed to best meet these.

Our tool will be a cross between a general programming environment such as ‘Microsoft Visual Studio’ [MSVS] or ‘Borland JBuilder’ [BORJB], and a UML development tool such as ‘Microsoft Visio’ [MSVIS] or ‘Rational Rose’ [RROSE]. The tool is similar to a general programming environment in the way that files will be created and a series of type / semantic checks will be carried out on them. It is similar to a UML development tool because obviously the user has the ability to construct class diagrams.

*Figure 4.3.* and *figure 4.4.* show the main visual components used in a typical UML development tool and programming environment.

A key component of most interfaces is a menu / tool bar. This allows the user to quickly access the various tools and capabilities offered by the system.

Both systems exhibit a type of directory tree which shows the structure of the current project. This type of component is essential in any sort of project development environment.

In each of the systems, most of the screen is taken up with a ‘main display area’. In the case of ‘Rational Rose’ this is the visual representation of the class diagram and in ‘Borland JBuilder’ this is the large text area where users can enter code.

Finally, a small ‘information area’ is displayed at the bottom of each of the systems. Both of these sections relay messages back to the user. With ‘Borland JBuilder’ this area also offers feedback on file compilations and execution.

Present in most development environments is the concept of a ‘properties table’. This displays various properties of the selected entity. The ‘Rational Rose’ property table cannot be seen in *figure 4.3.* as this is displayed in a separate window. However, when selecting the various UML constructs it would be very useful to see their properties directly on screen.

From this analysis we can say that our user interface will require the following core visual components:

1. Menu bar
2. Project / Directory tree
3. Main display area
4. Information / Message tabs
5. Properties table
figure 4.3. Main layout of Rational Rose

figure 4.4. Main layout of Borland JBuilder
4.6 Analysis of Visual Components

Here we describe the use of each of the components profiled in the previous section and how they relate to our system.

The ‘Menu bar’ will provide quick access to all the functionality of the system. Every available action / function will be accessed via this component.

The ‘Project / Directory Tree’ will display the contents of the current project i.e. all the UML construct instances which have been created.

The ‘Main display area’ will simply show a visual representation of the currently selected UML construct instance.

The ‘Information / Message tab’ will contain many tabs; each displaying a different category of information to the user i.e. type checking results, documentation etc. This will also double up as a text area where users can enter OCL constraints.

Finally, the ‘Properties Table’ will display the relevant properties of the currently selected UML construct instance.

4.7 GUI Layout Design

Now we have constructed a list of the required visual components we can look at how these should be laid out on screen. Figure 4.5 shows a typical layout for our interface which is a mix of ideas gained from the systems profiled in the previous sections.

![figure 4.5 Layout sketch for user interface]
4.8 GUI Implementation

The development will make use of Swing components, which are part of the Java Foundation Classes (JFC). The following table lists the swing components which will be required to implement the layout displayed in figure 4.5:

<table>
<thead>
<tr>
<th>Visual Component</th>
<th>Swing class equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu Bar</td>
<td>JMenuBar</td>
</tr>
<tr>
<td>Project / Directory Tree</td>
<td>JTree</td>
</tr>
<tr>
<td>Main Display Area</td>
<td>JPanel</td>
</tr>
<tr>
<td>Information / Message Tabs</td>
<td>JTabbedPane and JTextArea</td>
</tr>
<tr>
<td>Properties Table</td>
<td>JTable</td>
</tr>
</tbody>
</table>

The interface will also make use of other Swing classes including JFileChooser, JDialog, JColorChooser etc. for other smaller tasks.

4.9 Project / Directory Tree

This section profiles the design and implementation of the project / directory tree component and how it is used to represent a typical project.

Design

We have already concluded that the contents and structure of a typical project will be displayed in a tree like structure, but exactly how will we do this? We need to be able to group related constructs, this can be done using ‘folders’ and ‘sub-folders’ which are basic elements of a directory tree.

The first level of grouping will be at the project level. For every project we create we will create a parent folder which will contain all the constructs relating to this project.

The second level of grouping will be at the construct level. We have already decided that our tool will handle three types of construct, namely Classes, Interfaces and Associations. We will have a sub-folder for each of these types which will group all constructs of a particular type.

Finally, the lowest level of grouping will group individual elements of particular constructs. For example, we may define a class which has several methods and attributes, these will be grouped with that particular class.

Figure 4.6 displays an example project tree which has used this grouping methodology.
We will also need to consider issues of tree maintenance. When a user decides to add a new construct to the project this will need to be reflected in the project tree, and so an insert operation should be performed. Similarly, when a construct is removed, the project tree should also have this item deleted.

**Implementation**

The basic tree component is implemented in the library class *JTree*. This class contains a handle to the data which makes up the tree which resides in the library class *DefaultTreeModel*. This model class is then made up of a series of nodes which have a default implementation in the class *DefaultMutableTreeNode*. Figure 4.7 shows the relationship between these three classes.

![Diagram of the relationship between JTree, Default Tree Model, and Default Mutable Tree Node](image)

*Figure 4.7 Main library classes involved in the directory tree*

To create the first level of grouping i.e. at the project level, we simply make the root of the tree the ‘project folder’. For the second level of grouping i.e. at the construct level, we add three child nodes called ‘Classes’, ‘Interfaces’ and ‘Associations’. Successive additions of classes, interfaces and associations simply insert new instances into the relevant folders.

When we make an insertion we create a new instance of the particular construct and place this inside a *DefaultMutableTreeNode*. This node is inserted into the relevant folder. As a
result, when the node is later selected, the instance of the construct can be extracted from the node.

4.10 Properties Table

This section profiles the design and implementation of the properties table component and how it is used to display the properties of selected UML construct instances.

Design

In figure 4.6 we have an overall view of our project. Each of the constructs in the tree will have a series of properties which the user will want to modify. When the user clicks on a particular construct instance in the tree we should load the relevant data and display this in a custom properties table which the user can then modify.

When a particular construct instance is selected, the properties table will show particular properties of that instance according to the items displayed in the table in section 4.2. For example, if ‘ClassA’ is selected in figure 4.6, then a table containing Name, Extends, Abstract and Association Class properties will be displayed. Each of these properties will have a corresponding value which can be easily modified. Figure 4.8 shows the properties table for ‘ClassA’

This table is simply a high-level view of a portion of our project data structure. Updating the property table is equivalent to directly updating the data structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Extends</th>
<th>Abstract</th>
<th>Association Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.8 Properties table for the ‘Class’ construct

Implementation

The basic table component is implemented in the library class JTable. This class contains a handle to the data which makes up the table which resides in the class DefaultTableModel.

The table will need to show the properties of the currently selected construct instance in the project tree. When a new selection is made, information regarding the construct instance is extracted from the project tree node and the DefaultTableModel is made to point to this. Simply refreshing the table will cause it to show the new information. Any updates on the table will directly update the construct instance’s data structure.

4.11 Main Display Area

This section profiles the design and implementation of the main display area component and how it is used to render the selected construct instance.

Design

This is the main part of the screen and renders the currently selected construct instance. For each of the three UML constructs we offer, we will need to be able to produce a graphical
representation of them. We refer to [UML] to obtain template diagrams for each of our three constructs.

For a Class, the template diagram is shown in *figure 4.9*. This is comprised of three boxes. The first contains the Class name, the second contains a list of attributes and their types and the third box contains a list of methods, their parameter types and a return type.

```
ClassName

AttributeName1 : Type
AttributeName2 : Type
...

MethodName1( p1 : Type, ...) : Type
MethodName2( p1 : Type, ...) : Type
...
```

*figure 4.9. Template Class Diagram*

The template for an Interface can be seen in *figure 15*. This is similar to the Class template but differs in that it doesn’t offer attributes and that it has an explicit tag indicating that it is an interface.

```
<<Interface>> InterfaceName

MethodName1( p1 : Type, ...) : Type
MethodName2( p1 : Type, ...) : Type
...
```

*figure 4.10 Template Interface Diagram*

For Associations, we just show the two classes involved in the association and any labels and multiplicities belonging to it. The template can be seen in *figure 4.11*.

```
Left Type Name  Left multiplicity  Right multiplicity  Right Type Name
Left label  Right label
```

*figure 4.11 Template Association Diagram*
**Implementation**

This component is implemented using the `JPanel` class, it offers a `paint` method which is overridden with the rendering code.

The rendering algorithm detects which UML construct instance has been selected, gets a handle on its data structure and then displays it. Every time the currently selected construct instance is changed or a new construct instance is selected the display area is repainted.

For example, if `ClassA` is selected in *figure 4.6* then our rendering algorithm will produce the diagram in *figure 4.12*.

```
public void paint(Graphics g) {
    Object o = // get data for selected construct
    if (selectedConstruct instanceof Class)
        drawClass(o);
    if (selectedConstruct instanceof Interface)
        drawInterface(o);
    else
        drawAssociation(o);
}
```

The three `draw` methods each take advantage of the Java 2D API to conduct specific rendering to produce the diagrams.

**4.12 Information / Message Tabs**

This section profiles the design and implementation of the information / messages tab component. It also discusses how it can display certain messages to the user and how it is used to take documentation and OCL constraint input from the user.

**Design**
These series of tabs will have two purposes:

i. To display messages to the user regarding the results of various checks which will be conducted on the inputs.

ii. To allow the user to specify more information relating to the project.

The first purpose is dependent on what types of information we want to relay back to the user, and so these tabs will be dealt with in future chapters.

The second purpose deals with the fact that users will need an area on the screen to input extra relevant information. This extra information will come in two flavours, the first being OCL constraints and the second being documentation notes.

**Implementation**

We simply use a `JTabbedPane` component whereby we add the various tabs to this using the `addTab` method. So if we wanted to add two tabs which each contained text areas, one for entering documentation notes and the other for entering OCL constraints, then the following code would suffice:

```java
JTabbedPane tp = new JTabbedPane()
JTextArea jt_OCL = new JTextArea();
JTextArea jt_Doc = new JTextArea();

tp.addTab("Documentation", jt_Doc);
tp.addTab("OCL Constraints", jt_OCL);
```

### 4.13 File Persistence

**Design**

One essential requirement of a project development environment is offering the ability to save projects. In section 4.3 we defined a data structure which will hold the details of our project, but how can we best store this as a file?

One obvious way would be to simply write the data out to a text file, but this would lead to a complex process of reading the file back in when we wanted to re-open it. However, the Java language offers file persistence in the form of ‘Object Serialization’ whereby objects can be saved as a byte stream and then transferred to disk.

**Implementation**

When using Object Serialisation one requirement is that all saveable data structures must implement the `Serializable` Interface. We then modify the data structures in section 4.3 so that they meet this requirement.

File persistence is achieved in simply four steps. We first need to create a `FileOutputStream`, this represents the actual file which will store our project. We then use this to create an
ObjectOutputStream, our project data is then written to this and the stream is closed. The relevant code can be seen below:

```java
FileOutputStream fileOut =
    new FileOutputStream(fileName);
ObjectOutputStream objOut =
    new ObjectOutputStream(fileOut);
objOut.writeObject(projectData);
objOut.close();
```

4.14 Summary

Overall a Development Environment was produced which has the basic functionality of allowing the user to create UML class diagrams and to enter OCL constraints. This system will bring together all the tools which will be developed in future chapters so that the user can simply apply them to the input they’ve entered. The system offers additional functionality by way of allowing the user to create and save projects. Figure 4.13 shows a screen shot of the final system.

---

**figure 4.13 Screen shot of the Development Environment**
5. Class Diagram Validation

5.1 Overview

At this stage we will have a series of data structures which will hold the class diagram, these were designed in section 4.3. Validating the class diagram simply requires us to perform a number of basic checks on these data structures. It is important to note that these checks do not cover the full range of possible errors, subtle semantic inconsistencies may still be present however these inconsistencies should not greatly affect the ‘class diagram / OCL constraint’ integration process. We now profile the list of checks we need to conduct.

5.2 Class Validation

We need to check for the following possible errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name duplication</td>
<td>Do any classes share names</td>
</tr>
<tr>
<td>Non-existent super class</td>
<td>Does a class extend another class which does not exist</td>
</tr>
<tr>
<td>Cyclic inheritance</td>
<td>Is there a cycle in the inheritance structure</td>
</tr>
<tr>
<td>Incorrect class names</td>
<td>Class names cannot be equal to predefined basic types such as Integer, String etc.</td>
</tr>
</tbody>
</table>

**Name Duplication**

No two classes can share the same name.

We construct a list of all the class names contained in our model and simply check for duplications in this list.

**Non-existent super class**

Classes can only extend other classes which exist in the model.

We iterate through all the classes in the model and extract the ‘Extends’ property which tells us the name of the class which the current class extends. If this class is not null then we check for its existence in our model.

**Cyclic Inheritance**

There must be no cycles in the inheritance structure.

As in the previous check, for each class in the model we get hold of the ‘Extends’ property, this will give us the super class of the current class. We then navigate through the inheritance structure by accessing the ‘Extends’ property of the super class. This process continues until either the ‘Extends’ property resolves to null in which case we can assume that there are no cycles, or the ‘Extends’ property resolves to a class which we have previously navigated, which indicates a cycle.
**Incorrect Class Names**

A class name cannot be equal to one of the predefined basic types.

We build a list of all the class names and determine whether any of these are equal to the basic type names.

### 5.3 Class Attribute Validation

We need to check for the following possible errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name duplication</td>
<td>Do any attributes in a given class share names</td>
</tr>
<tr>
<td>Non-existent type</td>
<td>Is an attribute defined to be of a type which does not exist</td>
</tr>
</tbody>
</table>

**Name Duplication**

No two attributes, in a given class, can share names.

For each class in our model we construct a list of all the names of attributes which belong to that class and simply check for duplications in this list.

**Non-existent type**

An attribute must be of a type which exists in our model.

For each class in our model we iterate through it’s attributes. For each attribute, we extract the ‘Type’ property and check whether this exists in our model as either a class or an interface, or as one of the basic types.

### 5.4 Class Method Validation

We need to check for the following possible errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature duplication</td>
<td>Do any methods possess duplicate signatures</td>
</tr>
<tr>
<td>Non-existent types</td>
<td>Do any methods contain parameter types or have return types which do not exist</td>
</tr>
<tr>
<td>Incorrect method override</td>
<td>Do any methods incorrectly override inherited methods</td>
</tr>
</tbody>
</table>

**Signature Duplication**

No two methods, in a given class, can share signatures.

For each class in our model we iterate through it’s methods. For each method we extract it’s ‘Name’ and ‘Parameter Types’ properties which we can use to determine it’s signature. We
then check for duplications between this method signature and all other method signatures contained in the class.

**Non-existent types**

All types used in the definition of a method must exist in our model.

For each class in our model we iterate through it’s methods. For each method, we extract the ‘ReturnType’ and ‘Parameter Types’ properties and check whether these exist in our model as either classes or interfaces, or as one of the basic types.

**Incorrect method override**

If a class contains a method which has a signature which is duplicated in a super class, then both methods must have matching return types.

This implementation is a little more complicated than previous checks as it involves navigating the inheritance structure. For each class we need to build up a list of all the methods which it and it’s super classes contain. We then make sure that this list contains no methods which have duplicate signatures but differing return types.

### 5.5 Interface Validation

We need to check for the following possible errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name duplication</td>
<td>Do any interfaces share names</td>
</tr>
<tr>
<td>Incorrect interface name</td>
<td>Interface names cannot be equal to predefined types such as Integer, String etc.</td>
</tr>
</tbody>
</table>

The implementation for both of these checks is analogous to their Class counterparts.

### 5.6 Interface Method Validation

We need to check for the following possible errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature duplication</td>
<td>Do any methods possess duplicate signatures</td>
</tr>
<tr>
<td>Non-existent types</td>
<td>Do any methods contain parameter types or have return types which do not exist</td>
</tr>
</tbody>
</table>

The implementation for both of these checks in analogous to their Class Method counterparts.

### 5.7 Association Validation

We need to check for the following possible errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name duplication</td>
<td>Do any associations share names</td>
</tr>
</tbody>
</table>
### Non-existent association ends

Do the entities specified in an association actually exist?

### Non-existent association classes

Do all the specified association classes exist?

### Incorrect association class name format

Do all the specified association classes have a name which meets the required format?

### Name Duplication

Associations cannot share names.

The implementation is similar to the Name Duplication check of previous constructs.

### Non-existent association ends

The association ends must exist as either classes or interfaces in our model.

This amounts to iterating through all the associations, extracting the ‘Left Entity’ and ‘Right Entity’ properties, and checking if they exist as either classes or interfaces.

### Non-existent association classes

If an association class is used then this must exist in our model.

We iterate through the associations, extract the ‘AssociationClass’ property and check for its existence as a class in our model.

### Incorrect association class name format

In an OCL expression, if an association class is referred to by it’s name starting with an upper case later then this will resolve to that association class. If it is referred to by it’s name starting with a lower case later then this will resolve to a set of the association class. This means that we must make the restriction that all association class names should start with an upper case letter. The implementation of this is straightforward.

### 5.8 Integration with Development Environment

Linking this tool up to the Development Environment involves adding a menu item to the menu bar which will be used to invoke this tool, and adding an error message tab to the GUI which will be used for reporting model errors back to the user.
6. OCL Parser

6.1 Overview

Chapter 4 captured the OCL constraints and stored these in a single string. The fundamental aim of this section is to check that the contents of this string conform to the OCL grammar. This task is simplified by using a parser which also builds an Abstract Syntax Tree (AST) representation of the input. This can then be used by other processes for purposes such as type checking.

6.2 Design

The general methodology involves scanning the input and finding matches with this and grammar productions. Whenever part of the input is successfully consumed, a node will be created which is added to our partially built AST.

If the input conforms to the grammar then we will end up with an AST representation of the input. Otherwise we should report an error to the user, indicating the occurrence of the invalid input.

To explain the relevant components and classes which will be needed by our parser we first consider a simple grammar:

\[
\langle a \rangle ::= \text{ABC} \mid \text{DEF} \langle a \rangle
\]

Valid sentences which conform to this grammar will contain either ABC or a series of DEF followed by ABC:

\[
\begin{align*}
\text{ABC} \\
\text{DEF ABC} \\
\text{DEF DEF ABC} \\
\vdots
\end{align*}
\]

The first required component will be a ‘scanner’. The purpose of this will be to group characters into ‘words’ or ‘tokens’. With reference to our example grammar, if a scanner came up against the list of characters in figure 6.1 Then it should output the tokens shown in figure 6.2

\[
\begin{array}{ccccccc}
\text{D} & \text{E} & \text{F} & \text{A} & \text{B} & \text{C}
\end{array}
\]

*figure 6.1 Example list of characters*

\[
\begin{array}{cccc}
\text{DEF} & \text{ABC}
\end{array}
\]

*figure 6.2 Example set of tokens*
From this analysis it would be necessary to construct a ‘token’ class to represent the output of the scanner.

Once we have a steady stream of tokens we will need another component which will take these tokens, compare these to the grammar, and start building the AST for us. This will be the main component and we shall call it the ‘parser’.

In building the AST we will need to represent the nodes in the tree. Each grammar production will lead to a different type of node and so we will need a node class for each of the different productions and terminals. In our example grammar we would require the following node classes:

i. Node – Superclass of all nodes in the tree
    ii. a_Node – Represents the production \(<a>\)
    iii. ABC_Node – Represents the terminal \(ABC\)
    iv. DEF_Node - Represents the terminal \(DEF\)

So if we consider the tokens pictured in figure 6.2 then the resulting AST would be as displayed in figure 6.3

![Example AST](image)

We will not go into the internal details of the ‘parser’ as the JavaCC parser libraries will handle most of the work for us.

From this analysis, it is evident that we will need the following classes:

i. Scanner – To take the input and turn it into a series of tokens.
ii. Token – To represent the tokens which the scanner produces.
iii. Parser – To take the stream of tokens and produce the AST’s.
iv. Several Node classes – To represent each grammar production and terminal in the AST.

Finally, figure 6.4 shows how all these components fit together.
6.3 Implementation

Producing the basic components listed in the previous section is reasonably straightforward. We take the OCL grammar, which is expressed in Extended Backus-Naur Form, and run it through the JavaCC parser generator.

This process produces several key classes for us:

**Token Manager**

This corresponds to the scanner component which was listed in the previous section. It provides a key method with the following signature:

```java
public Token getNextToken()
```

Each call to this method extracts the next available token from the input, which is then fed into the parser.

**Token**

This represents a group of characters which are successfully extracted by the Token Manager. This offers several important attributes:

```java
public String image;
public Token next;
public int beginLine, beginColumn, endLine, endColumn;
```

The *image* property holds a string representation of the token. *next* is a reference to the next token in the input stream, this is useful if we want to access previous or successive tokens.

Finally, the *Line* and *Column* properties store the location of the token in the input stream, these are very useful for error reporting purposes.

**Parser**

This class does virtually the whole of the work for us. It offers the following constructor:

```java
public Parser(java.io.Reader stream)
```

We take our OCL input string and create a *Reader* object from it:

```java
StringReader oclTextString = new
```
We then use this to create an instance of the Parser:

```java
Parser p = new Parser(oclTextString);
```

The Parser also offers the method:

```java
public SimpleNode input()
```

Calling this method is equivalent to activating the Parser. It returns a `SimpleNode` object which represents the root of the newly created AST. This is demonstrated by the following line of code:

```java
SimpleNode s = p.input();
```

**Node Classes**

Approximately fifty node classes will be created for us, each corresponding to terminals and productions in the grammar. All these classes are subclasses of the superclass `SimpleNode` which itself implements the interface `Node`. This node hierarchy together with the basic node management methods can be seen in figure 6.5.
6.4 Error Reporting

Once a syntax error has been detected we should automatically report this back to the user. The main Parser class handles this for us. When a syntax error is found, a ParseException is raised, this is a subclass of the common Java class ‘Exception’.

When a syntax error is generated, the ParseException class holds the following information:

i.  `currentToken` – This is a reference to the token which violated the grammar.
ii. `expectedTokenSequences` – This is an array holding a series of tokens which the parser expected.

From the `currentToken` we can extract a string representation of the invalid token and also the position of this in the input. Figure 6.6 lists the basic structure of an error message reported by our parser.

```
“Encountered” InvalidToken “at line” lineNumber “, column” colNumber

“Was expecting:”
ExpectedTokenList
```

Figure 6.6. Error Message Format

6.5 Integration with Development Environment

Once the parser is complete we would like to link it up to the Development Environment. This involves adding a menu item to the menu bar which will be used to invoke the parser, and adding several message tabs which will be used for reporting information back to the user.

We will add two more message tabs to our Development Environment GUI. One will hold the error message output and the other will display the AST which results from a successful parse.
7. OCL Type Checker

7.1 Overview

We should now have the validated UML class diagram and OCL constraint input contained in a series of data structures. The real task of this module is to take both of these validated inputs and to check for type conformance between the two. In the process of checking for type conformance we will naturally check for the validity of all OCL references to the class diagram.

Initially we discuss the general methodology behind our type checker and give an example run through on a simple grammar. We then go onto create a type hierarchy involving all the types which will be used by the system.

We briefly cover the design and implementation of an ‘Environment’ class which will be used to store context information, and then we show how our type checker will interact with the UML class diagram.

We make a few adjustments to the main type checking classes to cater for the use of the ‘Environment’ class and the UML class diagram. Error reporting formats are briefly analysed together with the issue of integrating this tool with the overall Development Environment.

The bulk of the chapter then involves a detailed look at the implementation of the main type checking classes. With reference to the OCL grammar, we sequentially profile each of the grammar productions, discussing the main implementation issues.

7.2 General Methodology

The two main data structures which we will be working with are the AST representation of the OCL constraints and the series of UML construct instances. The process of type checking will involve a direct traversal of the AST. We start at the root node and determine it’s type by evaluating the type of it’s children. The types of it’s children are determined by the type of their children and so on until we reach the bottom of the tree where the types are propagated back up through the tree.

To explain this more clearly, we consider the following expression:

\[ 5 + 2.7 \times 3 \]

An AST representation of this expression can be seen in figure 7.1
Type checking the AST displayed in figure 7.1 will involve the following steps:

<table>
<thead>
<tr>
<th>No.</th>
<th>Action</th>
<th>‘+’ Type</th>
<th>‘5’ Type</th>
<th>‘*’ Type</th>
<th>‘2.7’ Type</th>
<th>‘3’ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Evaluate type of ‘+’ node</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Evaluate type of ‘+’ nodes children</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Evaluate type of ‘5’ node</td>
<td></td>
<td></td>
<td>Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Evaluate type of ‘*’ node</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Evaluate type of ‘*’ nodes children</td>
<td></td>
<td></td>
<td>Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Evaluate type of ‘2.7’ node</td>
<td></td>
<td></td>
<td>Integer</td>
<td>Real</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Evaluate type of ‘3’ node</td>
<td></td>
<td></td>
<td></td>
<td>Real</td>
<td>Integer</td>
</tr>
<tr>
<td>8</td>
<td>Determine type of ‘*’ node</td>
<td></td>
<td></td>
<td>Integer</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>9</td>
<td>Determine type of ‘+’ node</td>
<td>Real</td>
<td>Integer</td>
<td>Real</td>
<td>Real</td>
<td>Integer</td>
</tr>
</tbody>
</table>

We determine the type of the ‘+’ node by simply evaluating the types of it’s children. The ‘5’ node yields an Integer type. The type of the ‘*’ is determined by evaluating it’s children. The ‘2.7’ node yields a Real type and the ‘3’ node yields an Integer type. The types of ‘2.7’ and ‘3’ are then propagated back up to their parent, which is the ‘*’ node, this then evaluates to a type of Real. Finally, the type of the root node ‘+’ results from adding an Integer to a Real which gives us a Real.

The resulting type tree for this type checking example can be seen in figure 7.2
In section 3.2 it was decided that type checking a node would require a custom ‘Check’ class for each particular node. Each of these custom ‘Check’ classes would require a ‘Check’ method which would be responsible for all type checking of it’s corresponding node. For example, considering the AST in figure 7.1 We would have the following node classes:

i. Plus_Node  
ii. Multiply_Node  
iii. Number_Node

We would also have the corresponding type checking classes:

i. Check_Plus_Node  
ii. Check_Multiply_Node  
iii. Check_Number_Node

The typical structure of one of these ‘Check’ classes would be as follows:

```java
Class Check_Random_Node {
    public static Type Check(...) {
        // All type checking code for Random_Node
    }
}
```

Therefore, evaluating the type of a node amounts to invoking the ‘Check’ method associated with that node.

So referring back to the AST in figure 7.1 If the appropriate type checking code is produced for all three node’s, then evaluating the type of the AST would amount to the following call:

```
Check_Plus_node.check(..., Plus_Node,...)
```

*figure 7.2. Example type tree representation of 5 + 2.7 * 3*
The specific parameters with which each ‘Check’ method takes will be node dependent and will be discussed in future sections.

The previous chapter stated that our OCL parser will generate a series of node classes, corresponding to all the OCL grammar productions and terminals. The name given to each of these node classes takes the following format:

‘AST’ + name of production or terminal

So referring to the OCL grammar, the first 5 node classes are:

i. ASToclFile
ii. ASTpackage
iii. ASTendpackage
iv. ASTpackageName
v. ASToclExpressions

The name given to each of the corresponding type checking classes takes the following format:

‘Check’ + name of node class

7.3 Type Hierarchy

One of the most important issues in type checking is the construction of the type hierarchy. The types in the hierarchy will contain all the types recognisable in OCL. We refer to the OCL 1.4 specification [OCL1.4] to aid the design of this hierarchy. The overall type hierarchy can be seen in figure 7.3.

We initially create an abstract class Type which will be the super type of all types in the system. We extract the common operations needed by most types and place these into the Type class.

The type OclAny is considered to be the super type of all types in OCL apart from the collection types. Therefore we make Collection a separate subclass of Type. OclAny doesn’t specify any new methods and is merely in place to try and gain consistency with the OCL specification. It offers two subclasses, namely OclType which is the super type all types defined in our UML class diagram, and BasicType which is the super type of all basic types.

The collection types Collection, Set, Sequence and Bag simply represent the four types of collection allowed in OCL. Finally, we introduce Error and Statement which correspond to a type error and a well formed statement – these are considered as system types and do not specifically appear in OCL.
Another necessity in type checking is the use of an ‘Environment’. This holds information relating to the current context. In OCL there are three types of context information which we need to keep track of, we now discuss all three of these.

### Current Context Type

All OCL expressions are specified within a given context type. When we evaluate these expressions we will need access to this context type. For example, if we consider the following OCL constraint:
When we evaluate the item ‘attr1’, we will need to know what type this relates to. Therefore whenever we start type checking in a new context we will need to update the environment with the current context type.

In order to cater for this situation the environment will need to contain the following two methods:

```java
OclType getCurrentContextType();
void setCurrentContextType(OclType o);
```

**Pseudo Attributes and Operations**

OCL allows the specification of new attributes and operations on the class diagram. For example, assuming we had defined a class called ‘ClassA’ then the following constraints would add an attribute and an operation to ‘ClassA’:

```java
context ClassA def:
  let attr1 : Integer = 0
def:
  let op1(String s) : String = ‘Hello’
```

Therefore in future constraints we can refer to ‘attr1’ and ‘op1’ as members of ‘ClassA’.

These ‘Pseudo’ attributes and operations come in two flavours, those which are permanent i.e. after their specification they exist in all constraints, and those which are temporary i.e. after their specification they only exist in a small number of constraints. The environment will need the following four methods:

```java
ClassAttribute getPseudoAttribute(Type t, String name);
GeneralMethod getPseudoOperation(Type t, String name, List parTypes);
boolean setPseudoAttribute(String name, ClassAttribute ca, int duration);
boolean setPseudoOperation(String name, GeneralMethod cm, List parTypes, int duration);
```

**Type references**

There are several situations in OCL where references or names can be given to types. Future use of these references or names evaluate to the underlying type. For example, if we consider the following context declaration:

```java
context c : ClassA inv:
```
Then all constraints in the given context can refer to the reference ‘c’, which is of type ‘ClassA’. We can keep track of these references by adding the following two methods to the environment:

```java
Type getType(String name);
boolean putType(String name, Type t);
```

**Summary**

The class diagram in *figure 7.4* represents the environment which will be used with our type checker.

<table>
<thead>
<tr>
<th>&lt;interface&gt; Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OclType getCurrentContextType();</td>
</tr>
<tr>
<td>void setCurrentContextType(OclType o);</td>
</tr>
<tr>
<td>ClassAttribute getPseudoAttribute(Type t, String name);</td>
</tr>
<tr>
<td>GeneralMethod getPseudoOperation(Type t, String name, List parTypes);</td>
</tr>
<tr>
<td>boolean setPseudoAttribute(String name, ClassAttribute ca, int duration);</td>
</tr>
<tr>
<td>boolean setPseudoOperation(String name, GeneralMethod cm, List parTypes, int duration);</td>
</tr>
<tr>
<td>Type getType(String name);</td>
</tr>
<tr>
<td>boolean putType(String name, Type t, int type);</td>
</tr>
</tbody>
</table>

*figure 7.4 The ‘Environment’ Interface*

### 7.5 UML Class Diagram Information

Whereas the environment contains context specific information, our UML class diagram contains data which is accessible in all contexts. In other words, all constraints have access to the same class diagram data. When conducting our type checking we will need to get a handle on certain types in the class diagram. To enable this functionality we encapsulate the UML class diagram in a class which implements the interface ‘UMLModel’, this interface offers the following method:

```java
OclType getType(String name);
```

The ‘UMLModel’ interface can be seen in *figure 7.5*
More on ‘Check’ Classes

We stated in section 7.2 that each node in the AST representation of the OCL constraints will have a corresponding ‘Check’ class which will contain the relevant type checking code for that node. Each of these ‘Check’ classes will contain a type checking method which will be passed at least the following three parameters:

i. A reference to the node being type checked
ii. A reference to the environment
iii. A reference to the UML class diagram

In section 7.2 we listed the typical structure of one of these ‘Check’ classes. This structure can now be extended to include these parameters:

```java
class Check_Random_Node {
    public static Type Check(AST_Random_Node n,
                              Environment e,
                              UMLModel uml,
                              ...) {
        // All type checking code for Random_Node
    }
}
```

Error Reporting

A typical OCL constraint will be made up of a series of expressions which will have a specific type, and a series of statements which if evaluated correctly will have a type of ‘StatementType’. If the overall constraint returns a ‘StatementType’ then this indicates that the OCL constraint has been type checked successfully. If particular expressions return an ‘Errortype’ this indicates that a type / semantic error occurred. The ‘Errortype’ contains an attribute which holds the error message. When error reporting we access this error message property.

A typical OCL file will consist of many constraints, when we encounter an error in a constraint we make note of that error and start processing the next constraint. At the end of the type checking, if any errors occurred then we report back all of these, so that we only show at most one error in each constraint.

*Figure 7.6* shows the format of a typical error message generated by our type checker.
We display an error number which is a system identifier, and an error description which is a basic summary of the error. We then detail the occurrence of the type error in the input file, and specify several more pieces of error specific information.

### 7.8 Integration with Development Environment

As with the previous tools which were developed in chapters 5 and 6, we would like to link the type checker up to the Development Environment. This involves adding a menu item to the menu bar which will be used to invoke the type checker, and adding a message tab which will be used for reporting any type errors back to the user.

### 7.9 Clarifications

Due to the sheer size and complexity of this tool a few select areas will not be included. These areas are:

i. **Pseudo operations** – We will cover normal operations and pseudo attributes so the type checking of pseudo operations will be a cross between the two. Therefore in an attempt to reduce the complexity of the system and to save time we will not include pseudo operations.

ii. **Advanced associations** – We will cover basic labelled associations but areas such as unlabelled associations will not be included.

iii. **Enumerations** – Our Development environment does not cater for UML enumerations therefore the type checking of enumerations will not be included.

iv. **Declarator production** – This grammar production is sizeable and would require a complex implementation, therefore, this too will not be included.

Excluding the above areas will not have a great affect on the functionality of the type checker as we have only taken away a small portion of the overall task.

### 7.10 Implementation overview

We have defined the general architecture which will be used for implementing the type checker, now the main task is the creation of the type checking classes.

---

**figure 7.6 Error Message Format**

ErrorNumber“,,”ErrorDescription“:”

LineNumber, ColumnNumber

More Error Specific details ...

---

An Environment for integrating UML and OCL 47
The following sections profile the design and implementation of the type checking classes for the main grammar productions listed in the OCL grammar. We make an effort not to delve into a detailed implementation but to just highlight the main issues.

### 7.11 oclExpressions

**Production**

\[ \text{oclExpressions} ::= ( \text{constraint} )* \]

**Description**

This specifies zero or many constraints.

**Example**

The following example contains three constraint's:

```plaintext
context ClassA def:          // constraint 1
  let i : Integer = 10

context ClassB inv:          // constraint 2
  // expressions

context ClassC inv:          // constraint 3
  // expressions
```

**Analysis / Implementation**

Keypoints:

i. **Update Environment** - Each time we process a new constraint, the only information which will remain in the environment is any permanent pseudo attributes or operations which were defined in the previous constraint, by way of a def statement. Therefore we want to erase all other information held in the environment.

For each constraint we obtain, we simply update the environment and delegate the type checking responsibility to it. We return a StatementType if all constraint nodes evaluate to a StatementType otherwise we return an ErrorType.

### 7.12 constraint

**Production**

\[ \text{constraint} ::= \text{contextDeclaration} \]

\[ ( ( \text{“def”} \text{name} \text{“:”} \text{letExpression}* ) \]

\[ | \]

\[ ( \text{stereotype name} \text{“:”} \text{oclExpression} ) \]

\[ )+ \]
**Description**

This production represents a single OCL constraint. The `contextDeclaration` production allows the specification of the context type. The context can be any type that OCL is aware of.

The “def” side of the production allows one to specify a pseudo operation / attribute on the current context type.

The `stereotype` side of the production allows the user to specify an invariant, a pre-condition or a post-condition on the current context type.

**Example**

The following example specifies the context type as ‘ClassA’, uses a “def” to specify a pseudo attribute ‘i’ and adds an invariant:

```plaintext
context ClassA  // contextDeclaration
def: let i : Integer = 0  // ( "def" name? ":" letExpression* )
inv: attr1 > 10  // ( stereotype name? ":" oclExpression )
```

**Analysis / Implementation**

Keypoints:

i. **Evaluate contextDeclaration** – Initially we need to evaluate the `contextDeclaration` node. This will add the context type to the environment.

ii. **Evaluate “def” statements** – For each “def” statement we encounter we delegate the type checking responsibility to the `letExpression` node.

iii. **Evaluate stereotype statements** – For each `stereotype` statement we encounter we delegate the type checking responsibility to the `oclExpression` node.

If all the types in this `constraint` conform then we return a ‘StatementType’.

It must be noted that several semantic errors may occur at this stage, these all involve an incorrect usage of the `stereotype` statement and are detailed in chapter 8.

**7.13 contextDeclaration**

**Production**

```
contextDeclaration := “context”
( operationContext | classifierContext )
```

**Description**

This production allows the specification of two different kinds of context.

**Implementation**
We simply delegate the type checking responsibility to either the `operationContext` or `classifierContext` node, whichever is specified.

### 7.14 classifierContext

**Production**

```plaintext
classifierContext ::= ( name "::" name )
  | name
```

**Description**

This allows the specification of the name of the context type.

**Example**

The following example demonstrates the two types of `classifierContext` we may be faced with:

```plaintext
c : ClassA // ( name "::" name )
ClassA // name
```

**Analysis / Implementation**

The OCL specification allows a context type to be any random name, however we feel that we should make the restriction that the context type should be a type which exists in our UML class diagram. This is because all constraints are in relation to the class diagram and there would be no real need to specify a context type which did not exist. This restriction makes our type checker more strict and less prone to errors.

**Keypoints:**

i. **Type existence** - We need to check for the existence of the specified type in the class diagram, this would be ‘ClassA’ in our example.

ii. **Update environment** – If the context type is specified together with a reference i.e. the first alternative in our example, then we add the context type to the environment under the supplied reference. In our example the reference would be ‘c’. If no reference is supplied then we add it to the environment under the reference ‘self’.

iii. **Naming ambiguities** – If an explicit reference is supplied then we must check for naming ambiguities with this reference and other entities in OCL and the class diagram. So in our example we would need to check whether ‘c’ conflicted with any existing names.

If the `classifierContext` does identify a valid context type then we return a ‘StatementType’

### 7.15 operationContext
Production

operationContext := name "::" operationName
    "(" formalParameterList ")" ( "::" returnType )?

Description

This allows the specification of the name of the context type and the specification of one of the context type’s operations. Through using this type of context declaration we can specify constraints on operations.

Example

The following example specifies the context type as ‘ClassA’ together with an operation ‘meth1’.

```
ClassA :: meth1 // name "::" operationName
(i : Integer) // "(" formalParameterList ")"
: Integer // ( "::" returnType )
```

Analysis / Implementation

Keypoints:

i. **Type existence** - As with a classifierContext we need to initially check for the existence of the name node as a type in the UML class diagram i.e. does ‘ClassA’ exist, and then if successful we add it to the environment.

ii. **Evaluate operationName** – We need to extract the name of the operation from the operationName node.

iii. **Evaluate formalParameterList** – We evaluate the formalParameterList node. We need access to the parameter names and types which this holds so we also pass a couple of list objects into the formalParameterList node’s ‘Check’ method which extracts these for us.

iv. **Evaluate returnType** – We evaluate the returnType node, as a result we get access to it’s type.

v. **Operation existence** – Does the given operation exist. Now we have the operation name, the parameter types and the return type, does it exist as an operation on the context type. Referring back to our example, is ‘meth1’ an operation on ‘ClassA’.

vi. **Update environment** - If the context type does define such an operation then we add the parameter names to the environment and proceed.

If all the types in this operationContext conform then we return a ‘StatementType’.
7.16 formalParameterList

Production

formalParameterList := ( name “:” typeSpecifier
(“,,” name “:” typeSpecifier )* ) *

Description

This allows the specification of a list of parameter names and their types.

Example

The following example specifies three parameters with associated parameter types.

```
i : Integer       // name “:” typeSpecifier
, b : Boolean    // “,” name “:” typeSpecifier
, d : Date        // “,” name “:” typeSpecifier
```

Analysis / Implementation

Keypoints:

i. **Evaluate name’s** – We need to resolve all of the name nodes. These names may conflict with existing names e.g. they may conflict with UML class diagram types. Hence, for each parameter name we need to conduct a series of ambiguity checks.

ii. **Evaluate typeSpecifier’s** – We evaluate each of the typeSpecifier nodes. We need to check whether the associated types exist as either OCL types or types in the UML class diagram.

iii. **Extract names and types** – When we evaluate this node, as well as checking for type conformance we are also interested in the parameter names and types which it contains. After evaluating any specific name and typeSpecifier pairing, we add the resulting parameter name and parameter type to a list object which other code can access after the method completes.

If all the types in this formalParameterList conform then we return a ‘StatementType’.

7.17 typeSpecifier

Production

typeSpecifier := simpleTypeSpecifier | collectionType

Description

This allows the specification of a simple type or a collection type.
Implementation

We simply delegate the type checking responsibility to either the simpleTypeSpecifier or collectionType node, which ever is specified.

7.18  collectionType

Production

collectionType := collectionKind
   "(" simpleTypeSpecifier ")"

Description

This allows the specification of a collection and the element type of that collection.

Example

The following example creates a Set of Integers:

\[
\text{Set} ( \text{Integer} ) \quad \text{// collectionKind } "(" \text{simpleTypeSpecifier } "\)"
\]

Analysis / Implementation

Keypoints:

i. Evaluate collectionKind – We extract the type of the collection from the collectionKind node. In our example this would be ‘Set’.

ii. Evaluate simpleTypeSpecifier – We evaluate the simpleTypeSpecifier node. This must evaluate to a legal type. In our example this would be ‘Integer’.

If the previous evaluations are successful then we return a new instance of the collectionKind with the element type as simpleTypeSpecifier. So in our example we return a new instance of ‘Set’ with it’s element type equal to ‘Integer’.

7.19  oclExpression

Production

oclExpression := (letExpression * “in”) ? expression

Description

This allows the specification of a general expression together with an optional ‘let’ expression. The ‘let’ expression is used to define temporary pseudo members.

Example
The following example creates a temporary pseudo attribute ‘i’ which is then used in an expression which checks whether the value of ‘i’ is greater than 0.

```ocl
let i : Integer = 5 in // (letExpression * "in")
i > 0 // expression
```

**Analysis / Implementation**

Keypoints:

i. **Evaluate letExpression’s** – If any letExpression nodes are present then we evaluate them. This will update the environment with any pseudo members which the letExpression’s contain.

ii. **Evaluate expression** – All type checking responsibility is then delegated to the expression node which will dictate the overall return type of this oclExpression.

7.20 **returnType**

**Production**

```ocl
typeSpecifier
```

**Analysis / Implementation**

We delegate all type checking responsibility to the typeSpecifier node.

7.21 **expression**

**Production**

```ocl
logicalExpression
```

**Analysis / Implementation**

We delegate all type checking responsibility to the logicalExpression node.

7.22 **letExpression**

**Production**

```
letExpression := "let" name
   ( "(" formalParameterList ")" )?
   ( "." typeSpecifier )?
   "=" expression
```

**Description**

This allows the specification of a pseudo attribute / operation.
Example

The following example creates an attribute ‘i’ and an operation ‘addI’:

```java
// “let” name “:” typeSpecifier “=” expression
let i : Integer = 5

// “let” name “(” formalParameterList “)” “:” typeSpecifier “=” expression
let addI(x : Integer) : Integer = x + 10
```

Analysis / Implementation

Our type checker does not cover the use of pseudo operations therefore the implementation notes will only cover pseudo attributes.

Keypoints:

i. **Evaluate typeSpecifier** – We make the restriction that the specification of new attributes must come with a type. Through forcing the specification of an attribute type we avoid making unnecessary assumptions. We make note of the type which the `typeSpecifier` node evaluates to.

ii. **Check for Attribute Duplication** – Once we have the name of the attribute and the return type we can check whether the current context type defines a duplicate attribute, if so then the `letExpression` will be invalid. We also need to check for naming ambiguities between the attribute name and other entities in scope.

iii. **Evaluate expression** – We evaluate the `expression` node and make note of it’s type.

iv. **Type conformance** – The type of the new / pseudo attribute and the expression must conform otherwise we generate an error.

v. **Update environment** – If all the previous operations are successful then we add the attribute to the environment as a pseudo attribute on the current context type.

If all the types in this `letExpression` conform then we return a ‘StatementType’.

### 7.23 ifExpression

**Production**

```
ifExpression := “if” expression
    “then” expression
    “else” expression
    “end if”
```

**Description**

Allows the specification of an ‘If’ statement.

**Example**
The following example specifies a typical ifExpression:

```java
if true
then age = 18
else age = 21
end if
```

**Analysis / Implementation**

Keypoints:

i. **Evaluate “if” expression** – We evaluate the first expression, this must be of type Boolean. In our example, this would be equivalent to evaluating ‘true’.

ii. **Evaluate “then” expression** – We evaluate the second expression.

iii. **Evaluate “else” expression** – We evaluate the third expression.

iv. **Expression conformance** – The types resulting from the second and third expression nodes must produce types which conform to each other. In our example, the second expression ‘age = 18’ is of type Boolean and the third expression ‘age = 21’ is of type Boolean. These types conform to each other.

If the second and third expression’s conform to each other then the overall ifExpression evaluates to the type of either the second or third expression – whichever is more general. So our example would evaluate to a Boolean type.

### 7.24 logicalExpression

**Production**

```
logicalExpression := relationalExpression
                ( logicalOperator
                             relationalExpression
                      ) *
```

**Description**

Allows the specification of an expression with the use of logical operators.

**Example**

The following example specifies a typical logicalExpression:

```java
true
and false
or true
implies false
```
**Analysis / Implementation**

Keypoints:

i. **Evaluate relationalExpression’s** – We evaluate all `relationalExpression` nodes. If we have more than one `relationalExpression` node then they should all be of a Boolean type.

If all the types in this `logicalExpression` conform then the type we return depends on the number of `relationalExpression` nodes we receive. If we receive just one `relationalExpression` then we return the type of this, otherwise, if we receive more than one `relationalExpression` then we return a Boolean type.

### 7.25 relationalExpression

**Production**

```
relationalExpression := additiveExpression
    ( relationalOperator
        additiveExpression
    )?
```

**Description**

Allows the specification of an expression with the use of relational operators.

**Example**

The following example specifies a typical `relationalExpression`:

```
10 // additiveExpression
> 5 // relationalOperator additiveExpression
```

**Analysis / Implementation**

Keypoints:

i. **Evaluate additiveExpression’s** – We evaluate all the `additiveExpression` nodes.

ii. **Type conformance** – If we have two `additiveExpression` nodes then for a comparison operation, these nodes must of certain types - this is summarised in the table below.

<table>
<thead>
<tr>
<th>relationalOperator</th>
<th>Types it can be applied to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>Objects of any type</td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td>Objects of any type</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Integer or Real types</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Integer or Real types</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Integer or Real types</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Integer or Real types</td>
</tr>
</tbody>
</table>
If we have one additiveExpression then we simply return the type of this node. Otherwise, if all the types in this relationalExpression conform then we return a Boolean type. So in our example, ‘10’ is of type Integer and ‘5’ is of type Integer and the ‘>’ operator only accepts Integer or Real types, therefore our example is valid, and we return a Boolean type.

7.26 additiveExpression

Production

additiveExpression := multiplicativeExpression
   ( addOperator
    multiplicativeExpression
   ) *

Description

Allows the specification of an expression with the use of add operators.

Example

The following example specifies a typical additiveExpression:

```
10 // multiplicativeExpression
+ 5 // addOperator multiplicativeExpression
– 9 // addOperator multiplicativeExpression
```

Analysis / Implementation

Keypoints:

i. **Evaluate multiplicativeExpression’s** – We evaluate all multiplicativeExpression nodes. If we have more than one multiplicativeExpression node then they must all evaluate to a Real or Integer type.

If we have one multiplicativeExpression then we simply return the type of this node. If we have more than one multiplicativeExpression then the overall type depends on the individual multiplicativeExpression node types. So if any of the multiplicativeExpression nodes evaluate to a Real type then we return a Real type, otherwise we return an Integer type. In our example, we have three Integer’s therefore we return an Integer type.

7.27 multiplicativeExpression

Production

multiplicativeExpression := unaryExpression
   ( multiplyOperator
    unaryExpression
   ) *
**Description**

Allows the specification of an expression with the use of multiply operators.

**Example**

The following example specifies a typical `multiplicativeExpression`:

```plaintext
10 // unaryExpression
* 5 // multiplyOperator unaryExpression
/ 9 // multiplyOperator unaryExpression
```

**Analysis / Implementation**

Keypoints:

i. **Evaluate unaryExpression’s** – We evaluate all `unaryExpression` nodes. If we have more than one `unaryExpression` node then they all must evaluate to a Real or Integer type.

If we have one `unaryExpression` then we simply return the type of this node. If we have more than one `unaryExpression` then the overall type depends on the individual `unaryExpression` node types and the kinds of operators. So if any of the `unaryExpression` nodes evaluate to a Real type or we have a ‘/’ operator then we return a Real type, otherwise we return an Integer type. In our example we have three Integer’s, a ‘*’ operator and a ‘/’ operator. Due to the ‘/’ operator we return a Real type.

7.28 **unaryExpression**

**Production**

```plaintext
unaryExpression := ( unaryOperator
  postfixExpression
) | postfixExpression
```

**Description**

This allows the specification of an expression with the use of a unary operator.

**Example**

The following example specifies a typical `unaryExpression`:

```plaintext
not (10 > 5) // unaryOperator postfixExpression
```

**Analysis / Implementation**

Keypoints:
i. **Evaluate postfixExpression** – We evaluate the *postfixExpression* node. This will dictate the type of this *unaryExpression*.

ii. **Type conformance** – If a *unaryOperator* is specified then we must check that the operator is being used with a correct type. The table below shows the types which can be used with the different *unaryOperator*’s.

<table>
<thead>
<tr>
<th>Unary Operator</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Integer, Real</td>
</tr>
<tr>
<td>not</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

So our example is valid because the ‘not’ operator is being used with a *postfixExpression* which is of type Boolean.

The type of this *unaryExpression* will be equal to the type of the *postfixExpression*.

### 7.29 postfixExpression

**Production**

postfixExpression := primaryExpression
                 ( ( "." | ">" ) propertyCall ) *

**Description**

This allows the specification of an expression with a list of associated property calls.

**Example**

With reference to the UML class diagram in figure 7.7 the following example specifies a typical *postfixExpression*:

Person.age // primaryExpression "." propertyCall

```
Person

age : Integer

getAge() : Integer
```

*figure 7.7 ‘Person’ Class Diagram*

**Analysis / Implementation**

Keypoints:
i. **Evaluate primaryExpression** - We evaluate the primaryExpression node.

ii. **Evaluate propertyCall’s** – If any propertyCall nodes are encountered then these are evaluated from left to right.

When type checking each propertyCall node we must also supply the preceding type, so that we know what type we are invoking a property call on. In our example, from evaluating the primaryExpression we would obtain a ‘Person’ type. When evaluating the propertyCall ‘age’ we supply the type checking code the ‘Person’ type. The propertyCall would then return an Integer type which would be the type of the whole postfixExpression.

### 7.30 primaryExpression

#### Production

```plaintext
primaryExpression := literalCollection \\
| literal \\
| propertyCall \\
| "(" expression ")" \\
| ifExpression
```

#### Description

This allows the specification of five different types of expressions ( the node names should indicate what they are ).

#### Analysis / Implementation

We simply delegate the type checking responsibility to the node that we encounter.

### 7.31 propertyCallParameters

#### Production

```plaintext
propertyCallParameters := "(" ( declarator ) ? \\
| ( actualParameterList ) ? \\
| ")"
```

#### Description

Allows the specification of a declarator and/or a list of parameters inside mandatory brackets.

#### Example

```plaintext
( i | i + 10 ) // "(" declarator actualParameterList ")"
```

#### Analysis / Implementation
Our type checker does not cover the use of a declarator therefore the implementation notes will only cover an actualParameterList.

Keypoints:

i. **Evaluate actualParameterList** – Evaluating this propertyCallParameters node simply amounts to type checking the actualParameterList.

ii. **Extract parameter types** - As well as checking for type conformance, ‘calling’ code will be interested in the types of the parameters contained in the actualParameterList. Therefore we accept a list object which is also passed to the actualParameterList type checking code, this list is used to store the parameter types, so that other code can access them.

If all the types in this actualParameterList conform then we return a ‘StatementType’.

### 7.32 literal

**Production**

```plaintext
literal := string
       | number
       | enumLiteral
```

**Description**

Allows the specification of a string, a number or an enumeration literal.

**Example**

```
'hello world'        // string
10000                // number
```

**Analysis / Implementation**

Our type checker does not cover the use of enumerations therefore the implementation of type checking a enumLiteral will not be covered.

If we receive a string node then we return a String type. If we receive a number node then we must check whether it contains a decimal point. If it does then we return a Real type, otherwise we return an Integer type.

### 7.33 simpleTypeSpecifier

**Production**

```plaintext
simpleTypeSpecifier := pathName
```

**Description**

Allows the specification of a type.
**Analysis / Implementation**

We extract the name from the `pathname` node and check whether this name identifies a valid type. If it does identify a valid type then we return this.

### 7.34 literalCollection

**Production**

```
literalCollection := collectionKind "{"
                     (collectionItem
                      ("," collectionItem )*)?
                     "}"
```

**Description**

Allows the specification of a collection and it’s elements.

**Example**

```
Set { 1, 2.1 } // collectionKind "{"
```

**Analysis / Implementation**

Keypoints:

i. **Evaluate CollectionKind** – We extract the type of the collection from the `collectionKind` node. In our example this would be ‘Set’.

ii. **Evaluate CollectionItem’s** – We evaluate all the `collectionItem` nodes. If any of these nodes resolve to a collection then these are flattened. For example if a `literalCollection` is equal to Set {1, Set{2, 3}} then the Set{2, 3} `collectionItem` will be flattened so that overall we have Set {1, 2, 3}.

iii. **Type conformance** – All `collectionItem` nodes must conform to one general type. In our example we have a Real and an Integer, as Integer is a subclass of Real, both of these types conform to Real.

The overall type will be dictated by the `collectionKind` and the general type which all `collectionItem` nodes conform to. In our example we have a ‘Set’ and all the `collectionItem` nodes conform to a Real type therefore we would return a Set(Real).

### 7.35 collectionItem

**Production**

```
collectionItem := expression (".." expression)?
```
Description

Allows the specification of a collection element or a range of collection elements.

Example

(1 + 2) .. (5 + 6)       // expression “..” expression

Analysis / Implementation

Keypoints:

i. Evaluate expression’s – We evaluate each of the expression nodes we obtain.

If we receive one expression node then we simply delegate all type checking responsibility to this node. If we receive two expression nodes then they both must evaluate to an Integer type, in which case we return an Integer type.

7.36 propertyCall

Production

propertyCall := pathName
  (timeExpression) ?
  (qualifiers) ?
  (propertyCallParameters) ?

Description

Allows the specification of a property call. This is the most important grammar production.

Example

If we consider the UML class diagram in figure 7.8, typical examples would be:

name       // pathName
name@pre   // pathName timeExpression
ggetName(1) // pathName propertyCallParameters

figure 7.8. ‘Company’ Class Diagram
Analysis / Implementation

There are only five types of `propertyCall` which are semantically correct. For the time being we focus on these five types, but in chapter 8 we discuss this issue in more detail. The five valid types of `propertyCall` are:

i. `pathName`
ii. `pathName timeExpression`
iii. `pathName propertyCallParameters`
iv. `pathName timeExpression propertyCallParameters`
v. `pathName qualifiers`

When type checking this `propertyCall` node we are supplied with the preceding type (that’s if there is one). The next few sections deal with type checking the five kinds of `propertyCall` listed above. Instead of detailing a thorough implementation we just highlight the main issues.

Case 1: `pathName / pathName timeExpression`

The `timeExpression` shouldn’t make any difference to our type checks. However there are several semantic issues involving the `timeExpression` which will be covered later.

If we consider a random constraint such as the following:

```context classA inv:
  property > 10
```

Then ‘property’ could resolve to any of the following entities:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type name</td>
<td>The direct name of a type in the UML class diagram</td>
</tr>
<tr>
<td>A reference</td>
<td>A reference to a type such as the keyword ‘self’, this is a reference to the context type</td>
</tr>
<tr>
<td>A Boolean literal</td>
<td>i.e. ‘true’ or ‘false’</td>
</tr>
<tr>
<td>An attribute</td>
<td>An attribute on the context type</td>
</tr>
<tr>
<td>A pseudo attribute</td>
<td>A pseudo attribute on the context type</td>
</tr>
<tr>
<td>An association</td>
<td>An association involving the context type</td>
</tr>
<tr>
<td>‘result’</td>
<td>Could be equal to ‘result’ which has a special meaning in post-conditions</td>
</tr>
</tbody>
</table>

We check for the resolution of the `pathName` to each of the seven entities listed above. If it resolves to more than one entity then we raise an ambiguity error. If it resolves to none of the entities then the property cannot exist. If it only resolves to one of the entities then we obtain the type of that entity.

We now change the example constraint so that it includes a preceding type:

```context classA inv:
  PrecedingType.property > 10
```
Now the ‘property’ can only resolve to one of three things:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>An attribute</td>
<td>An attribute on the preceding type</td>
</tr>
<tr>
<td>A pseudo attribute</td>
<td>A pseudo attribute on the preceding type</td>
</tr>
<tr>
<td>An association</td>
<td>An association involving the preceding type</td>
</tr>
</tbody>
</table>

As before, we check for the resolution of the `pathName` to each of the entities listed above and return the appropriate type.

**Case 2 : `pathName propertyCallParameters / pathName timeExpression propertyCallParameters`**

Again, the `timeExpression` shouldn’t make any difference to our type checks. This type of property call can only lead to an operation. We obtain the operation name from the `pathName` node and the parameter types from the `propertyCallParameters` node. We check for the existence of the operation on either the context type or it’s preceding type (if it exists). If the operation exists then we return the return type of the operation.

**Case 3 : `pathName qualifiers`**

This type of property call can only lead to a qualified association. We obtain the association name from the `pathName` node and the qualifier information from the `qualifiers` node. We check for the existence of the association on either the context type or it’s preceding type (if it exists). If the association exists then the resulting type of this `propertyCall` will be dictated by the qualifiers.

### 7.37 qualifiers

#### Production

`qualifiers := "[" actualParameterList "]"`

#### Description

Allows the specification of a series of qualifiers.

#### Example

```
[dog, 1 + 2, 'a']       // "[" actualParameterList "]"
```

#### Analysis / Implementation

The implementation is the same as for our implementation of a `propertyCallParameters` node.
pathName := name ( "::" name )*  

**Description**  
Allows the fully qualified specification of a name.  

**Example**  
```plaintext
Name1 :: Name2  // name :: name
```

**Analysis / Implementation**  
Keypoints:  

i. **Evaluate name’s** – Instead of returning a type, we evaluate all the `name` nodes and return the fully qualified name which they hold.

7.39 **actualParameterList**  

**Production**  
```plaintext
actualParameterList := expression ("," expression)*
```

**Description**  
Allows the specification of a series of parameter expressions.  

**Example**  
```plaintext
dog, 1 + 2  // expression "," expression
```

**Analysis / Implementation**  
Keypoints:  

i. **Evaluate expression’s** – We evaluate all of the `expression` nodes.  

ii. **Extract type information** – Other type checking methods will be interested in the types of the parameters so we store these in a list object which other code / methods can access.

If all the types in this `actualParameterList` conform then we return a ‘StatementType’.
8. Semantic Issues

8.1 Overview

The previous chapter concerned itself with the ‘type issues’ involved in the integration of UML and OCL. This chapter deals with OCL statements and expressions which are syntactically valid but semantically flawed. For each case we detail the semantic issues, provide examples of their occurrence and then if possible, either offer an improvement to the grammar or detail how our type checker could be improved to deal with the issue.

8.2 Time Expression

In a postcondition, the expression can refer to two sets of values for each property of an object [OCL1.4]:

i. The value of a property at the start of the operation or method.

ii. The value of a property upon completion of the operation or method.

To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword ‘@pre’ [OCL1.4], this is referred to as a timeExpression.

If we consider the UML class diagram illustrated in figure 8.1 the following constraint correctly exploits the usage of a timeExpression:

context Person :: birthdayHappens()
post: age = age@pre + 1

This example states that after the completion of the operation ‘birthdayHappens()’, the ‘age’ attribute will be incremented by one.

Such functionality is useful but can be used incorrectly. The grammar allows a timeExpression to be used in any type of constraint. As a result we can end up with a timeExpression being used in an invariant or pre-condition, in which the notion of a previous value does not occur. The following example, again referencing figure 8.1 illustrates this issue:

context Person :: birthdayHappens()
pre: age = age@pre + 1
In this example the previous value of the attribute ‘age’ is being referred to in a precondition which is semantically incorrect.

We can extend our type checker by making note of the current constraint type in the environment i.e. whether we are dealing with an invariant or a precondition etc. If we processed a timeExpression we would then make sure that the current constraint type is a postcondition, if not we would raise an error.

8.3 Types of Constraint

The three main types of constraints are preconditions, postconditions and invariants. Invariants are specified on classes, and preconditions and postconditions are specified on operations. However the OCL grammar allows the specification of all three of these types of constraint on both classes and operations. With reference to figure 8.1 the following examples are syntactically correct but semantically flawed:

```
context Person :: birthdayHappens() : void
  inv: age = 50

context Person
  pre: age = 10
```

We can extend our type checker to cope with this without too much trouble. We know the following:

i. The type of constraint is stored in a stereotype node

ii. A context declaration specifies either a class or a class and an operation

iii. If the context declaration specifies a class then it will be stored using a classifierContext node

iv. If the context declaration specifies a class and an operation then it will be stored using an operationContext node

So if we make note of the kind of context declaration we have used by storing it in the environment, when we process the stereotype node we can make sure that invalid combinations of context declarations and constraints do not occur.

With reference to the earlier examples, in the first case the context declaration is stored using an operationContext node. This means that we will be specifying constraints on a class operation. When we process the stereotype node, we find that we have an invariant which cannot be used with an operation therefore we error.

In the second case, the context declaration is stored using a classifierContext node. This means that we will be specifying constraints on classes. When we process the stereotype node, we find that we have a precondition which cannot be used with just a class, therefore we error.
8.4 Unspecified Return Type

A \textit{letExpression} allows the definition of a new attribute or operation. It allows the attribute type or the operation return type to be optional. We find ourselves in the position that attributes can be specified without a type and operations can be specified without a return type. The following example demonstrates this:

\begin{verbatim}
let i = 10                 // An attribute
let random() = 5000       // An operation
\end{verbatim}

We can determine the attribute type or operation return type by evaluating the type of the accompanying expression and assuming that this will be the same. So in our examples, both right hand expressions are of Integer type, so the attribute ‘i’ and operation ‘random()’ will have Integer types. However, making such assumptions is dangerous, and a simple improvement to the grammar would be to make the attribute type or the operation return type compulsory. In the OCL grammar this corresponds to making the \textit{typeSpecifier} compulsory in the \textit{letExpression}.

8.5 Invalid property calls

The \textit{propertyCall} production is as follows:

\begin{verbatim}
propertyCall := pathName
   ( timeExpression ) ?
   ( qualifiers ) ?
   ( propertyCallParameters ) ?
\end{verbatim}

So we could have the following eight combinations of \textit{propertyCall}:

i. \textit{pathName}
ii. \textit{pathName} \textit{timeExpression}
iii. \textit{pathName} \textit{qualifiers}
iv. \textit{pathName} \textit{propertyCallParameters}
v. \textit{pathName} \textit{timeExpression} \textit{qualifiers}
vi. \textit{pathName} \textit{timeExpression} \textit{propertyCallParameters}
 vii. \textit{pathName} \textit{qualifiers} \textit{propertyCallParameters}
  viii. \textit{pathName} \textit{timeExpression} \textit{qualifiers} \textit{propertyCallParameters}

The pairing of \textit{timeExpression} \textit{qualifiers} has no meaning therefore combinations (v.) and (viii.) are invalid. Also, the pairing of \textit{qualifiers} \textit{propertyCallParameters} has no meaning which rules out combination (vii).

If we receive any of the eight combinations listed earlier then these will pass a syntax check. However, they may be semantically invalid, and so to catch any invalid \textit{propertyCall}’s we examine the structure of the \textit{propertyCall} and simply check for the three invalid combinations previously stated.
8.6 Division by zero

A *multiplicativeExpression* allows expressions to be created which involve the division operator ‘/’. Expressions may be specified whereby the right hand side of the ‘/’ operator evaluates to 0:

\[
25 / (10 - 10)
\]

This expression will obviously be mathematically invalid. We could get round this problem by evaluating all mathematical expressions which involved a ‘/’ operator. The downside of this is that it would lead to a complex implementation.

8.7 Incorrect usage of ‘self’ and ‘result’

The words ‘self’ and ‘result’ both have special meanings in OCL. ‘self’ is used as a reference to the current context type and ‘result’ is used to refer to the result of an operation in a postcondition. Despite their importance, these keywords are not explicitly represented in the grammar and so there is nothing to stop the use of these keywords as names for references to other entities. For example, parameter names or pseudo attributes could be named ‘self’ or ‘result’. With reference to *figure 8.3* the following examples demonstrate these two situations:

```
context Person :: meth1(self:Integer) : Integer
pre:  self > 10

context Person
def:  let result : Integer = 10
```

```

<table>
<thead>
<tr>
<th>ClassA</th>
</tr>
</thead>
<tbody>
<tr>
<td>meth1(self:Integer) : Integer</td>
</tr>
<tr>
<td>meth2(r:Real) : Integer</td>
</tr>
</tbody>
</table>

*figure 8.3. ‘ClassA’ Class Diagram*

Our type checker copes with these situations by explicitly checking all parameter names and pseudo attribute names. We also make the restriction that type, attribute and association names cannot be equal to ‘self’ or ‘result’ so as to avoid ambiguities.

8.8 Contradictory Constraints

A more advanced semantic issue relates to the scenario where constraints are specified which contradict each other. We consider the following two constraints, both in relation to *figure 8.3*:

```
context Person :: meth2(r:Real) : Integer
post: result > 10

context Person inv:
```

An Environment for integrating UML and OCL
meth2(10) < 0

The first constraint specifies a postcondition which states that the result of the ‘meth2’ operation should yield a value greater than ‘10’. Whereas the second constraint specifies an invariant which says that the result of the ‘meth2’ operation with input ‘10’ should be less than ‘0’. These constraints are mutually exclusive, they cannot both be satisfied at the same time. A solution to this problem would be complex and would inevitably involve an expression evaluator to reduce the constraints to their lowest form.
9. Evaluation

9.1 Overview

So far this project has detailed the design and implementation of four key tools: the ‘Development Environment’, the ‘Class Diagram Validator’, the ‘OCL Parser’ and the ‘OCL Type Checker’. In order to evaluate these tools this chapter contains a detailed case study which documents the integration of an example UML class diagram with several OCL constraints. The example class diagram which we will be working with is taken directly from the OCL 1.4 specification – this is illustrated in figure 9.1. We also conduct a performance test which gives a rough approximation to the system response time.

![Class Diagram](image)

*figure 9.1 Official OCL 1.4 Specification example Class Diagram*

9.2 Case Study

Initially, we will recreate the class diagram (figure 9.1) in our Development Environment and then run this through the Class Diagram Validator to check for any inconsistencies with the UML specification. We will then profile the integration of this class diagram with three example OCL constraints. For each constraint we focus on, we will pass them through the OCL Parser which will check for syntax errors and then we will check for type conformance between each constraint and the class diagram. To show off the type checking capabilities...
of the system we will purposely produce several type errors in each of the example constraints, demonstrating how we detect these.

**Class Diagram Input**

We note that the Development Environment was not designed to cater for qualified associations and enumerations hence the ‘accountNumber’ qualifier and the ‘Sex’ enumeration will not be included in this case study.

The easiest way to enter the class diagram information is to specify one class at a time together with it’s associated attributes and operations. Once we have specified the necessary classes we can then focus on inputting the association information.

We now detail the creation of an example class. To create the ‘Person’ class we select the ‘Add Class’ option from the ‘Classes’ menu bar, this is shown in figure 9.2 This inserts a new class construct into our project.

![Figure 9.2 The Classes menu bar](image)

*figure 9.2 The Classes menu bar*

*figure 9.3 shows the new look of our project directory tree after the class creation.

![Figure 9.3 The Project Directory tree after the class creation](image)

*figure 9.3 The Project Directory tree after the class creation*

The class name defaults to ‘Class1’. We alter the name to ‘Person’ by modifying the class ‘Name’ property in the properties table. The updated properties table can be seen in figure 9.4

![Figure 9.4 The class properties table after the name change](image)

*figure 9.4 The class properties table after the name change*

Now the ‘Person’ class is created we need to add it’s various attributes and operations. Adding an attribute can be achieved in a similar fashion. To add the ‘isMarried’ attribute we first need to create a class attribute instance. We highlight the ‘Person’ class in the project
tree and select the ‘Add Attribute’ option from the ‘Classes’ menu bar. This inserts a new class attribute construct into our project under the ‘Person’ class. The updated project tree can be seen in figure 9.5.

![Image 1](image1.png)

**figure 9.5 The Project Directory tree after the attribute creation**

The attribute name defaults to ‘Attribute1’. We alter the name to ‘isMarried’ by modifying the attribute ‘Name’ property in the properties table. We also modify the ‘Type’ property by changing it to ‘Boolean’. The updated properties table can be seen in figure 9.6.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>isMarried</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

![Image 2](image2.png)

**figure 9.6 The attribute properties table after the name and type change**

All the attributes in the ‘Person’ class can be added in a similar way. However, the ‘sex’ attribute is excluded as this refers to an enumeration.

Adding operations is also very straightforward. To add the ‘income’ operation we first need to create a class method instance. We highlight the ‘Person’ class in the project tree and select the ‘Add Method’ option from the ‘Classes’ menu bar. This inserts a new class method construct into our project under the ‘Person’ class. The updated project tree can be seen in figure 9.7.

![Image 3](image3.png)

**figure 9.7. The Project Directory tree after the method creation**

The method defaults to the name of ‘Method1’. We alter the name to ‘income’ by modifying the method ‘Name’ property in the properties table. We also modify the ‘Return Type’
property by changing it to ‘Integer’, and we alter the ‘Par 1 Type’ property to ‘Date’. The updated properties table can be seen in figure 9.8

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Abstract</th>
<th>Static</th>
<th>Return Type</th>
<th>Par 1 Type</th>
<th>Par 2 Type</th>
<th>Par 3 Type</th>
<th>Par 4 Type</th>
<th>Par 5 Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*figure 9.8 The method properties table after the property changes*

A drawback of the method properties table is the fact that we can only specify up to five parameters. This decision was largely due to the width of the properties table i.e. the more properties we allow the wider it gets. A definite improvement would be to redesign the properties table such that the properties were listed in a vertical fashion. Microsoft Visual Studio [MSVS] offers a development environment which contains such a properties table, this can be seen in figure 9.9. Listing the properties would enable us to display a virtually unlimited amount of properties, therefore we could have a virtually unlimited number of parameter types.

![Properties table](image)

*figure 9.9 Microsoft Visual Studio Properties table – properties are listed vertically*

The creation process for the ‘Bank’, ‘Company’, ‘Marriage’ and ‘Job’ classes is straightforward and is analogous to that of the ‘Person’ class. The only difference is that for the two association classes ‘Marriage’ and ‘Job’ we must set the ‘Association Class’ property to true. If we refer back to the class properties table in figure 9.4 this would be equivalent to ticking the checkbox under the ‘Association Class’ column.

The final stage of inputting our class diagram is the specification of the associations. Our example class diagram in figure 9.1 contains four associations. Two of these are basic labelled associations and the other two involve association classes. We now describe the process of adding a basic labelled association.
To add the ‘manager – managedCompanies’ association between ‘Person’ and ‘Customer’ we select the ‘Add Association’ option from the ‘Associations’ menu bar, this is shown in figure 9.10. This inserts a new association construct into our project.

![Associations menu bar](image)

*figure 9.10 The Associations menu bar*

The association defaults to the name of ‘Association1’. We alter the name to ‘Ass1’ by modifying the association ‘Name’ property in the properties table. The rest of the details describing the association are also entered into the properties table. The updated properties table can be seen in figure 9.11.

<table>
<thead>
<tr>
<th>Name</th>
<th>Assoc. Class</th>
<th>Left entity</th>
<th>Left label</th>
<th>Left mult.</th>
<th>Right entity</th>
<th>Right label</th>
<th>Right mult.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ass1</td>
<td>Person</td>
<td>manager</td>
<td></td>
<td></td>
<td>Company</td>
<td>managedComp</td>
<td>0..*</td>
</tr>
</tbody>
</table>

*figure 9.11 The association properties table after the property changes*

The other three associations can be added in a similar way. The two associations which exhibit an association class additionally need to have their association class specified under the ‘Assoc. Class’ property.

The project can be saved using the ‘Save Project’ menu item. The entire project tree can be seen in figure 9.12.
Class Diagram Validation

Now that the class diagram has been entered into our system we want to check for consistency between this and the UML specification. We run this through our Class Diagram Validator which conducts a series of checks over the input, this was detailed in chapter 5. After selecting the ‘Check UML Validity’ option from the ‘Tools’ menu bar, as shown in figure 9.13 the output is displayed at the bottom of the screen. This output can be seen in figure 9.14.
The output from the Class Diagram Validation tells us that we have four errors, which all relate to the ‘date’ type. We have used ‘date’ throughout our model but have failed to define it. To correct this error, we create a new class called ‘Date’. Running the Class Diagram Validator again gives us no errors which implies that the class diagram is now valid.

Note: Class names are type insensitive, therefore ‘Date’, ‘DATE’ or ‘date’ all represent the same class.

OCL Parsing and Type Checking

We now profile the parsing and type checking of three OCL constraints in relation to our example class diagram. All constraints are specified inside the package ‘dave’.

**Constraint 1**

```ocl
class Company
  inv:
    self.numberOfEmployees > 50
```

This constraint states that the ‘numberOfEmployees’ attribute in the ‘Company’ class should be greater than ‘50’. This is entered into the OCL text area and the ‘Check OCL
Syntax’ option is selected from the ‘Tools’ menu bar. This constraint passes the syntax check and the AST in figure 9.15 is created.

We change the constraint so that the ‘context’ token is taken out:
Checking the validity of this constraint now leads to the following error message:

```
Checking OCL Syntax...

Errors encountered:

Encountered "Company" at line 1, column 1.
Was expecting one of:
"endpackage" ...
"context" ...
":" ...

#CONTEXT: 'Company'
context Company inv:
  self.numberOfEmployees > 50
```

The error correctly identifies the fact that we are missing the ‘context’ token. It also reports the occurrence of the error and what tokens would have been accepted instead of the ‘Company’ token. However, in larger constraints it may be awkward for the user to identify the occurrence of the invalid token in the actual constraint (by using the ‘line number’ and ‘column number’ information). So it may be useful to instead highlight the invalid token in the constraint.

The ‘context’ token is re-entered so that the syntax is once again correct. We now decide to check for type conformance between this constraint and our class diagram. We select the ‘Check UML vs OCL’ option from the ‘Tools’ menu bar which invokes the type checker. As expected, no type errors are present in the constraint.

We change the constraint so that we use a String in the comparison:

```
context Company inv:
  self.numberOfEmployees > 'Hello World'
```

When we invoke the type checker we obtain the following error message:

```
ERROR #36 - Incorrect Types in relational Expression, should both be either Integer or Real types

#LINE: 2. COLUMN: 9
#SUB-EXPRESSIONS: 'self . numberOfEmployees > 'Hello World''
#SUB-EXPRESSION TYPES: 'Integer > String'
in:
#CONTEXT: 'Company'
```

The type checker has picked out the type error. It indicates the occurrence of the invalid expressions and details the types of these expressions.

We change the constraint so that we use an attribute which has not been defined in the class diagram, we call this attribute ‘random’:

```
context Company inv:
  self.random > 50
```
Invoking the type checker supplies us with the following error message:

![Error message]

The type checker correctly identifies the error. This time the type checker can’t resolve the ‘random’ property and so it flags an error; this could be identified as more of a semantic error than a type error.

**Constraint 2**

```plaintext
context Company inv:
    self.random > 50
```

We now profile a more complicated constraint. The constraint specifies a postcondition on the ‘Person’ class. It initially declares a pseudo attribute and then uses this in an ‘If’ expression. As before, this is entered into the OCL text area and the ‘Check OCL Syntax’ option is selected from the ‘Tools’ menu bar. This constraint passes the syntax check.

Checking for type conformance also leads to an error-free constraint. We change the constraint so that we incorrectly specify the return type of the ‘income’ operation:
Type checking this new constraint against our class diagram leads to the following error message:

```
ERROR #08 - Operation does not exist,
#LINE: 1, COLUMN: 21
#OPERATION: 'income'
in:
#OPERATION CONTEXT: 'Person :: income ( d : Date ) : void'
```

The type checker cannot find an ‘income’ operation on the ‘Person’ class which has identical parameter and return types as specified. This may be deemed more of a semantic error than a type error.

We readjust the constraint and this time put an error in the pseudo attribute declaration; we change the type of ‘isHappy’ to a Real type:

```
context Person :: income(d:Date) : Integer post:
    let isHappy : Boolean = (age > 18) in
    if (isHappy and (firstName = 'Dave')) then
        result = 10000
    else
        result = 100
    endif
```

Running the type checker generates the following error message:

```
ERROR #26 - Type mismatch,
#LINE: 3, COLUMN: 9
#LEFT EXPRESSION: 'isHappy'
#LEFT TYPE: 'Real'
#RIGHT EXPRESSION: '( age > 18 )'
#RIGHT TYPE: 'Boolean'
in:
#LET STATEMENT: 'let isHappy : Real = ( age > 18 )'
#CONTEXT: 'Person'
```
The type checker detects a mismatch in types between the pseudo attribute ‘isHappy’ and it’s equivalent expression.

Finally, we create a type error in the ‘If’ expression. We put ‘firstName.concat(lastName)’ inside the ‘If’ test:

```plaintext
context Person :: income(d:Date) : Integer post:
  let isHappy : Boolean = (age > 18) in
  if (isHappy and (firstName.concat(lastName))) then
    result = 100000
  else
    result = 100
  endif
```

As expected the type checker generates the following error message:

```plaintext
ERROR #30 - All sub-expressions in a logical Expression must be of type Boolean,
#LINE: 5, COLUMN: 13
#SUB-EXPRESSION: '(firstName . concat ( lastName ))'
#SUB-EXPRESSION TYPE: 'String'
in:
#EXPRESSION: '(isHappy and (firstName . concat (lastName )))'
#CONTEXT: 'Person'
```

The error message tells us that ‘(firstName . concat (lastName ))’ resolves to a String and that this type cannot be used inside a logical expression.

**Constraint 3**

```plaintext
context c : Company inv:
  Person.income( c.manager.wife.husband.birthDate ) +
  Bank.customer.employer->size() = 100
```

This constraint doesn’t have much meaning but truly exploits the functionality of the type checker as it involves full navigation of the example class diagram. The constraint successfully passes both a syntax check and a type check.

We change the constraint so that we access the ‘isUnemployed’ attribute of ‘Person’ rather than the ‘birthDate’ attribute:

```plaintext
context c : Company inv:
  Person.income( c.manager.wife.husband.isUnemployed ) +
  Bank.customer.employer->size() = 100
```
Invoking the type checker on this constraint yields the following error message:

```
ERROR #87 - No matching operation,

#LINE: 3, COLUMN: 16
#PROPERTY CALL: 'income ( c.manager.wife.husband.isUnemployed )'
#PARAMETER TYPES: 'income(Boolean)'
#ON TYPE: 'Person'
in:
#EXPRESSION: 'Person.income ( c.manager.wife.husband.
            isUnemployed ) + Bank.customer.employer->size() = 100'
#CONTEXT: 'Company'
```

The type checker resolves ‘c.manager.wife.husband.isUnemployed’ to be of type Boolean and then tries to find an ‘income(Boolean)’ operation on the ‘Person’ type. It can’t find such an operation so an error is signalled.

Instead of accessing the ‘employer’ association label on ‘Person’ we can access the ‘Marriage’ association class:

```
context c : Company inv:
    Person.income( c.manager.wife.husband.birthDate ) +
    Bank.customer.marriage = 100
```

Type checking this constraint leads to the following error message:

```
ERROR #45 - Incompatible Types in additive Expression, should contain either
Integer or Real types

#LINE: 3, COLUMN: 9
#SUB-EXPRESSION: 'Bank.customer.marriage'
#SUB-EXPRESSION TYPE: 'Set(Marriage)'
in:
#EXPRESSION: 'Person.income(c.manager.wife.husband.birthDate) +
    Bank.customer.marriage = 100'
#CONTEXT: 'Company'
```

The type checker has resolved the ‘Bank.customer.marriage’ expression to be of type ‘Set(Marriage)’ which is incompatible with an additive expression.

**Summary**

This case study has demonstrated how the tools developed within this project can be used to integrate OCL constraints and UML class diagrams. We evaluated the use of our tools on an example class diagram and several OCL constraints, and found that numerous type errors were correctly detected. However, the tools did have some drawbacks, chapter 10 lists the necessary enhancements and improvements that would be needed to make this project more complete.
9.3 Performance Assessment

This section will evaluate the system response time i.e. after entering the relevant class diagram and constraints, how long will the system take to complete the consistency checks.

We choose the UML class diagram to be identical to the one used in the case study, this is shown in figure 9.1. We will compare the system response time against varying amounts of constraints i.e. how will the system response time vary when it processes an increasing number of constraints. This just gives us a rough idea as to the number of OCL constraints the system can handle before it’s performance becomes unacceptable.

We pick what we consider to be an ‘average set of constraints’ i.e. A typical set of constraints which the system will need to process. We decide that this set of constraints will contain the three constraints which we processed in the earlier case study. As a reminder they are:

```context Company inv: // NO. 1
  self.numberOfEmployees > 50
context Person :: income(d:Date) : Integer post: // NO. 2
  let isHappy : Boolean = (age > 18) in
  if (isHappy and (firstName = 'Dave')) then
    result = 10000
  else
    result = 100
  endif
context c : Company inv: // NO. 3
  Person.income( c.manager.wife.husband.isUnemployed ) +
  Bank.customer.employer->size() = 100```

We then duplicate this ‘average set of constraints’ n times and feed them into the system, each time we do this we make note of the processing time. Running varying amounts of these constraints on a Pentium 200Mhz produces the following table:

<table>
<thead>
<tr>
<th>Number Of Constraints</th>
<th>Response Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.461</td>
</tr>
<tr>
<td>300</td>
<td>1.124</td>
</tr>
<tr>
<td>500</td>
<td>2.475</td>
</tr>
<tr>
<td>700</td>
<td>4.284</td>
</tr>
<tr>
<td>900</td>
<td>14.4</td>
</tr>
</tbody>
</table>

The accompanying graph can be seen in figure 9.16. As the number of constraints approaches 900 the system response time starts rapidly increasing. If our system was to have more than 1000 constraints defined then it would be well worth investigating the optimisation of the type checking algorithms. For this system, we would not envisage more than a few hundred constraints being defined which would mean that the system response time would only be a couple of seconds in the worst case – this is well acceptable.
It must be noted that if we conducted the same test on a different set of constraints then we may experience different results. This is due to the fact that different constraints are processed in different ways. However, this performance test still gives us a rough gauge as to the speed and performance of our system on an ‘average set of constraints’.

![System Response Time Vs No. of Constraints](image)

*Figure 9.16 Performance Graph*

9.4 Final Evaluation

**Meeting the Objectives**

The objectives of the project were to produce a tool which would allow users to:

1. create UML class diagrams
2. add OCL constraints
3. facilitate a series of consistency checks across both of the inputs

This project was able to provide a Development Environment which allows users to graphically construct UML class diagrams. Allowing the user to create class diagrams in a graphical way makes the input process very easy and efficient. However, in comparison to existing UML development software such as ‘Rational Rose’ our Development Environment is less flexible and less complete. Our Development Environment does meet the objective of allowing the user to create class diagrams, but only offers the user a small number of UML constructs which puts a limitation on the range of class diagrams which can be created.
The addition of OCL constraints is catered for by the Development Environment and proved to be very simple. We simply provide the user with a text area where they are free to enter such constraints.

The issue of consistency checks proved to be the most complex task. Before we could conduct any ‘consistency checks’ we needed to validate the given input.

Validating the class diagram was difficult, the UML specification is very long and precise and so producing a fully functional class diagram validator which is 100% consistent with the UML specification would be a project in itself. The Class Diagram Validator which was produced handles most class diagram errors but does fail to detect a few advanced cases. For example, if we have an abstract class which defines a series of abstract methods then we do not check that non-abstract subclasses of this class actually provide definitions for each of these methods – but essentially this has little affect on the OCL/UML integration process.

Validating the OCL input lead to the creation of the OCL Parser. This tool takes advantage of a series of ‘tried-and-tested’ parser libraries and offers 100% grammar conformance. This area of the project is fully complete.

We managed to identify ‘consistency checks’ as being mainly a matter of type checking. This project produced an OCL Type Checker which conducts approximately 95% of all possible checks. Certain type checks were left out either due to time considerations or their sheer complexity, inevitably this has led to a decrease in system usability. As well as type checking, the case study showed that the Type Checker does also conduct several semantic checks on its input. These semantic checks are essential to the implementation of the type checker.

We exceeded our objectives by detailing several semantic issues (chapter 8) and discussing how our type checker could be adapted to cater for them.
10. Conclusions

10.1 Overview

This chapter covers the possible improvements and enhancements to the tools developed in this project, and then finishes with a final word on the overall project result.

10.2 Future work

Improvements

The following is a list of improvements that can be made to our system:

i. We mentioned in the case study that we couldn’t deal with qualified associations and enumerations as the Development Environment does not recognise them. Adding these two constructs would definitely add to the usability and applicability of the system.

ii. As the properties table displays all the properties in a row this puts a limit on the number of properties which can be seen on screen for each UML construct. If we instead list the properties from top to bottom we could fit more properties on screen. This was discussed in detail in the case study and would benefit the Class Method construct.

iii. When we discussed the design and implementation of the Type Checker we noted that it did not conduct all possible type checks. In particular, it does miss out the processing of the declarator production which can be useful when dealing with collections. Improving the Type Checker to cater for the declarator production and the other type checks which it misses would increase the functionality of the Type Checker.

iv. The Class Diagram Validator does not deal with all error cases. Some advanced error cases are not catered for so an obvious improvement would be the modification of the Class Diagram Validator so that it is 100% consistent with the UML specification.

v. As the user heavily interacts with the Development Environment it would be useful in putting this through a usability test so that the GUI can be fine tuned specifically to the users needs.

Enhancements

The following is a list of enhancements, which are new ideas that can be added to improve the functionality of the system:

i. We could add another level of OCL validation by conducting a series of semantic checks on the constraints. In chapter 8 we detailed several semantic issues and how they could be detected, the implementation of these could be included with the type checker.
ii. Our system creates an AST representation of the OCL constraints, other tools could be created which work on this AST which provide other useful features. For example, another tool which is currently being developed in the department provides a mapping between OCL and Prolog. This mapping tool could be adapted so that it works on the OCL AST. As a result, any OCL constraints which were entered into the system could be mapped into Prolog.

10.3 Overall

Most of the tools which have been developed in this project are based on existing ideas, however the integration of all of these tools to produce an overall ‘UML and OCL’ driven design environment is one of the first of it’s kind. Essentially, this set of tools can be used to aid the software development process, and in particular, to strengthen the validity and accuracy of the design stage, hence increasing the robustness and validity of resulting software.
## X1. Bibliography

| AKE     | David Akehurst, *OCL Expression Evaluator* (Application)  
|         | www.cs.ukc.ac.uk/people/staff/dha/xInterests/UMLandOCL/ |
| ARGO    | Open Source, *Argo – Object Orientated design tool* (Application)  
|         | argouml.tigris.org/ |
| BORJB   | Borland, *JBuilder 6.0* (Application) |
| FF2000  | Frank Finger, *Design and Implementation of a Modular OCL Compiler* |
| GOF     | Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides,  
|         | *Design Patterns: Elements of Reusable Object-Orientated Software*  
|         | Addison-Wesley, 1994 |
| JAVACC  | JavaCC – *Java Compiler Compiler*,  
|         | www.webgain.com/products/java_cc/ |
| MSVIS   | Microsoft Corporation, *MS Visio* (Application) |
| MSVS    | Microsoft Corporation, *MS Visual Studio* (Application) |
| OCL1.4  | Object Management Group, *Object Constraint Language 1.4 specification* |
| RBCOM   | Roger Bailey, *Second Year Compilers course* |
| RROSE   | Rational Software Corporation, *Rational Rose* (Application) |
| SABCC   | SableCC - *The Sable Research Group's Compiler Compiler*,  
|         | www.sablecc.org |
| UML1.4  | Object Management Group, *Unified Modelling Language 1.4 specification* |
| UUML    | Perdita Stevens, Rob Pooley. *Using UML*. Addison-Wesley, 2000 |
| XER     | Xerces - *Java Parser* (Application),  
|         | xml.apache.org/xerces-j/ |
X2. OCL 1.4 Grammar

oclFile := ( "package" packageName
  oclExpressions
  "endpackage"
 )+

packageName := pathName

oclExpressions := ( constraint )*

constraint := contextDeclaration
  ( ( "def" name? "=": letExpression* )
  | ( stereotype name? "=": oclExpression)
 )+

contextDeclaration := "context"
  ( operationContext | classifierContext )

classifierContext := ( name "=" name )
  | name

operationContext := name "::" operationName
  "(" formalParameterList ")"
  ( ":=" returnType )?

stereotype := ( "pre" | "post" | "inv" )

operationName := name | "=" | "+" | "-" | "<" | "<=" | ">=" | ">" | "implies" | "not" | "or" | "xor" | "and"

formalParameterList := ( name "=" typeSpecifier
  ("," name "=" typeSpecifier )*
 )?

typeSpecifier := simpleTypeSpecifier
  | collectionType

collectionType := collectionKind
  "(" simpleTypeSpecifier ")"

oclExpression := (letExpression* "in")? expression

returnType := typeSpecifier

expression := logicalExpression

letExpression := "let" name
  ( "(" formalParameterList ")"
  | "=": typeSpecifier )?
  "=": expression

ifExpression := "if" expression
  "then" expression
  "else" expression
  "endif"

logicalExpression := relationalExpression
  ( logicalOperator
    logicalExpression
  )

An Environment for integrating UML and OCL
relationalExpression := additiveExpression 
    ( relationalOperator 
        additiveExpression 
    )? 

additiveExpression := multiplicativeExpression 
    ( addOperator 
        multiplicativeExpression 
    )* 

multiplicativeExpression := unaryExpression 
    ( multiplyOperator 
        unaryExpression 
    )* 

unaryExpression := ( unaryOperator 
    postfixExpression 
  ) | postfixExpression 

postfixExpression := primaryExpression 
    ( ( "." | ":=" )propertyCall )* 

primaryExpression := literalCollection 
    literal 
    propertyCall 
    "(" expression ")" 
    ifExpression 

propertyCallParameters := "(" ( declarator )? 
    ( actualParameterList )? ")" 

literal := string 
    number 
    enumLiteral 

enumLiteral := name ":=" name ( ":=" name )* 

simpleTypeSpecifier := pathName 

literalCollection := collectionKind "{" 
    ( collectionItem 
        ( "," collectionItem )* 
    )? 
    "}" 

collectionItem := expression ( "." expression )? 

propertyCall := pathName 
    ( timeExpression )? 
    ( qualifiers )? 
    ( propertyCallParameters )? 

qualifiers := "{" ( actualParameterList ")" 

declarator := name ( "," name )* 
    ( ":" simpleTypeSpecifier )? 
    ( ":" name ":=" typeSpecifier "=" expression 
    )? 

pathName := name ( ":=" name )* 

timeExpression := @"pre" 

actualParameterList := expression ( "," expression)*
logicalOperator := "and" | "or" | "xor" | "implies"

collectionKind := "Set" | "Bag" | "Sequence" | "Collection"

relationalOperator := "=" | ">" | ">=" | "<=" | "<>"

addOperator := "+" | "-"

multiplyOperator := "*" | "/"

unaryOperator := "-" | "not"

typeName := charForNameTop charForName*

ame := charForNameTop charForName*

charForNameTop := /* Characters except inhibitedChar and ["0"-"9"]; the available characters shall be determined by the tool implementers ultimately.*/

charForName := /* Characters except inhibitedChar; the available characters shall be determined by the tool implementers ultimately.*/

inhibitedChar := ";"|"|"|"|"|"|"|"|"|"|"

number := ["0"-"9"] {["0"-"9"]}* ( ("." ["0"-"9"] {["0"-"9"]})* )? ( ("e" | "E") ( "+" | "-" )? ["0"-"9"] {["0"-"9"]}* )?

string := ""|

("""

) | ["\n","\t","\b","\r","\f","\",",","""] |

{["0"-"7"]} |

{["0"-"7"]} |

) |

) |

)*

* *
Note: The Case Study in chapter 9 gives a detailed walk through for a typical session.

All the files relating to this project can be obtained from the following URL:

www.doc.ic.ac.uk/~dms99/project/

Once this ‘project’ folder has been copied to the appropriate directory, type the following:

D:\project> cd src
D:\project\src> java -cp . project.part1.Main

This will load the main Development Environment.

**How do I create a UML Class Diagram?**

Creating instances of the various UML constructs on offer is explained below:

<table>
<thead>
<tr>
<th>UML Construct</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Select ‘Add Class’ from the ‘Classes’ menu bar</td>
</tr>
<tr>
<td>Class Attribute</td>
<td>Highlight relevant Class and select ‘Add Attribute’ from the ‘Classes’ menu bar</td>
</tr>
<tr>
<td>Class Method</td>
<td>Highlight relevant Class and select ‘Add Method’ from the ‘Classes’ menu bar</td>
</tr>
<tr>
<td>Interface</td>
<td>Select ‘Add Interface’ from the ‘Interfaces’ menu bar</td>
</tr>
<tr>
<td>Interfaces Method</td>
<td>Highlight relevant Interface and select ‘Add Method’ from the ‘Interfaces’ menu bar</td>
</tr>
<tr>
<td>Association</td>
<td>Select ‘Add Association’ from the ‘Associations’ menu bar</td>
</tr>
</tbody>
</table>

**How do I validate the UML Class Diagram?**

Simply select ‘Check UML Validity’ from the ‘Tools’ menu bar.

**How do I add OCL Constraints?**

These need to be specified in the ‘OCL Constraints’ tab at the bottom of the screen.

**How do I validate the OCL Constraints?**

Simply select ‘Check OCL Syntax’ from the ‘Tools’ menu bar.

**How do I perform Consistency Checking?**

Simply select ‘Check UML vs OCL’ from the ‘Tools’ menu bar.
How can I print out the Class Diagrams?

The tool does not provide any printing functionality, however, pressing ‘Alt-PrintScreen’ will produce a screen dump which can be pasted into a typical graphics package and printed from there.