Swansea University E-Theses

Software frameworks for embedding formal specifications and documentation in object oriented programming languages.

Tao, Shu

How to cite:

Tao, Shu (2007) *Software frameworks for embedding formal specifications and documentation in object oriented programming languages..* thesis, Swansea University.
http://cronfa.swan.ac.uk/Record/cronfa42302

Use policy:

This item is brought to you by Swansea University. Any person downloading material is agreeing to abide by the terms of the repository licence: copies of full text items may be used or reproduced in any format or medium, without prior permission for personal research or study, educational or non-commercial purposes only. The copyright for any work remains with the original author unless otherwise specified. The full-text must not be sold in any format or medium without the formal permission of the copyright holder. Permission for multiple reproductions should be obtained from the original author.

Authors are personally responsible for adhering to copyright and publisher restrictions when uploading content to the repository.

Please link to the metadata record in the Swansea University repository, Cronfa (link given in the citation reference above.)

http://www.swansea.ac.uk/library/researchsupport/ris-support/

Software Frameworks for Embedding Formal Specifications and Documentation in Object Oriented Programming Languages

Shu Tao

Tutor: Dr. Neal A. Harman

Submitted to the University of Wales Swansea in fulfilment of the requirements for the Degree of Master of Philosophy of Computer Science

July 2007

ProQuest Number: 10798010

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10798010

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Declaration

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

01/19/2009

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Also those other sources are acknowledged by footnotes giving explicit references and that a bibliography is appended.

01/19/2009

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

01/19/2009

Dedication

To My Father and Mother!

Acknowledgements

To ensure that this thesis is accurate, complete and targeted at the aim of the research, Dr. Neal A. Harman has not only reviewed this thesis and made extensive suggestions, but continually pointed me back at the practical problems in the progress of the research. A special thanks to Dr. Neal A. Harman for all his much-appreciated hard work and guidance. Thanks also to all my friends who had supported me in these three years.

Summary

This thesis forms part of a project on formally specifying and documenting Object-Oriented programming languages. In particular, this thesis investigates the construction of a generic software framework which can provide a formal specification and documentation model for C# classes including important concepts such as inheritance.

Object-oriented languages are promoted for their ability to provide a modular approach to programming, allowing programmers to design classes that perform a common set of tasks that can then be easily reused and expanded upon. The *syntax* of the public interface of a class itself can and has been documented in a reasonably formal way, the *semantics* is usually defined simply with natural language. In order to try and solve this problem, we use self-defined utilities to include formal semantics to improve the quality of the documentation. By that means, we can also extract a complete formal specification.

Formal specifications and documentations have been proved to be a practical way of modelling and testing specific systems when applied to certain kinds of problem: mainly *safety critical applications,* where the need for specific mathematical expertise is outweighed by the consequences of system failure. What is lacking at the moment is there are no industrial toolsets to make development using formal specifications faster and easier. It is hard for people to understand and use specifications if they do not know any specification language. At the moment it is generally necessary to have substantial mathematical knowledge of the underlying logics used by the languages. This project is about building a generic toolset which is possible to make the task of producing formal specifications in a reasonable and easier way.

Content

Dedication 3
Acknowledgements 4
Summary 5
Content 6
Chapter 1 Introduction 8
1.1 A Brief Look at Examples 14
1.2 Overview of Thesis 19
Chapter 2 Literature Review and Overview of Chosen Technologies 20
2.1 Object-Oriented Programming and C# 20
2.1.1 The Current State of Object-Oriented Programming 20
2.1.2 C# 21
2.2 Documentation 21
2.3 Formal Specifications 23
2.4 Attributes and Reflection in C# 27
2.4.1 Attributes 28
2.4.2 Reflection 30
2.5 XML 31
2.5.1 Easy Data Transfer and Exchange 31
2.5.2 Customizing Markup Languages 32
2.5.3 Self-Documenting Data 33
2.5.4 Structured and integrated Data 34
2.5.5 The Five Pre-existing Entity References 34
2.6 XSL 35
2.7 Maude 36
Chapter 3 The Overview of the Research 38
3.1 Motivation 38
3.2 Bringing in the XML Specification 41
3.3 The Executable Algebraic Specification 43
3.4 A More Generic Approach 45
3.5 Overview of the Work in this Thesis 49
Chapter 4 Embedding Generic Formal Specifications and Documentation 51
4.1 Transforming the EADF into C# Code 51
4.1.1 Defining the Attributes in XML 51
4.1.2 Handling Constructors: An Extended XML Format 59
4.1.3 The Attributes Generator 63
4.2 Modelling the ESC into XML Class Specification 64

4.2.1 The Structure and the Syntax of the Class Reader 65
4.2.2 Modelling the ESC into XML Class Specification 73
4.3 From XML Class Specification to Maude Specification 81
4.3.1 The Translator 81
4.3.2 Class and its Name 82
4.3.3 Fields 82
4.3.4 Constructors 84
4.3.5 Methods 85
4.3.6 Equations 88
4.3.7 Sorts, Subsorts and Hidden operators 90
4.3.8 Inheritance 90
4.4 The Future Development of the Current System 93
Chapter 5 Examples of Class Specification 94
5.1 ArrayList 94
5.2 The Shapes Example 103
Chapter 6 Future Development 114
6.1 A Prospect for a Universal Structure 114
6.2 A Prototype of the Universal Object-Orient Class Model 117
6.3 An Alternative Formalism: Pre-Post Conditions 121
6.4 Further Work 123
6.5 Conclusion 124
Chapter 7 Bibliography 126

Chapter 1 Introduction

This thesis forms part of a project on formally specifying and documenting Object-Oriented programming languages. In particular, this thesis investigates the construction of a generic software framework which can provide a formal specification and documentation model for C# classes including important concepts such as inheritance. The eventual aim of the work here is: to define an XML sub-language permits a wide range of formal specification methods to be declaratively defined and embedded within a range of different object-oriented programming languages; and to build a software system that actually generates the corresponding specification/ documentation. By building an XML-sub-language we are able to define the behavour we require *declaratively* - in a style similar to, say, XAML [31] which permits declarative declaration of user interfaces. Declarative programming in this style only requires the user to state *what* is required, and eliminates the need to define *how* it is achieved. This is a very substantial project, and it is not possible to complete it within a single thesis. Here, we concentrate on a single (algebraic) specification/documentation style (though we do partially explore another, based on pre and post-conditions), and a single language (C#); the XML sub-language is only partially defined, and the corresponding implementation is not complete.

This research builds on the work of J. Biddle [1] by generalising it so that specification techniques other than algebraic can be used and implementing it for the C# programming languages ([1] uses Java). Also [1] concentrates on the underlying theoretical model of embedded algebraic specification: this thesis is concerned with the more practical problem of building a software framework for embedding specification (including algebraic specification, which forms our main example.) We will introduce the concept of an *Embedded Specification*, ES, which we will use as an intermediary stage in the specification process. The ES is intended to show what we consider to be the key information in the specification and documentation of an object-oriented class. *Metadata and datatypes* [17] are the fundamental blocks of every object-oriented program, and they can be hard to extract from the program due to their various forms. Here we combine the ES and XML to represent this data in a more generic way, which are more readable and easier to access as well as being less language-specific. In order to illustrate the generic tools we have built, we will illustrate them with an example based on equations and Maude [40] (which is an executable algebraic specification language, see details in Chapter 2.7).

We will also in this thesis address the following smaller aims:

1. To automate as much as possible of the generation of the formal specification and documentation.
2. To allow a way of embedding the formal specification and documentation in a simple format within the C# class itself.

Object-oriented languages are promoted for their ability to provide a modular approach to programming, allowing programmers to design classes that perform a common set of tasks that can then be easily reused and expanded upon. A fundamental concept is that all another programmer would need to know is the public interface of a class and they would then be able to utilise the class within their own programs. However although the *syntax* of the public interface of a class itself can and has been documented in a reasonably formal way, the *semantics* is usually defined simply with natural language. C# in particular offers an XML documentation comment style with three slash marks (///) [2]. These comments can be extracted by associated software tools. Similarly, Java offers a method of embedding automatically extractable documentation in comments [77]. The Visual Studio [43] editor can recognize C# documentation comments and helps format them properly, and the C# compiler can process these comments into an XML file. The following example is a simple C# class with XML documentation comments [44]:

```
using System;

/// <summary>
/// A simple C# Class about Person</summary>
/// <remarks>
/// This class is created for document testing </remarks>
class Person
{
    /// <summary>
    /// Initialise the myName property</summary>
    private string myName ="N/A";

    /// <summary>
    /// Initialise the myAge property</summary>
    private int myAge = 0;

    /// <summary>
    /// The class constructor. </summary>
    public Person()
    { }

    /// <summary>
    /// Name property </summary>
```

```
/// <value>
/// the value of Name can be set by users</value>
public string Name
{
      get
      {
            return myName;
      }
      set
      {
            myName = value;
      }
}

/// <summary>
/// Age property </summary>
/// <value>
/// the value of Age can be set by users</value>
public int Age
{
      get
      {
            return myAge;
      }
      set
      {
            myAge = value;
      }
}

/// <summary>
/// override the method ToString() in C#</summary>
/// <returns>
/// which returns a string contains both Name and Age.</returns>
public override string ToString()
{
      return "Name = " + Name + ", Age = " + Age;
}

 /// <summary>
/// Main code for input and output</summary>
public static void Main()
{
      Console.WriteLine("Simple Properties");
```

```
            Person person = new Person();
            Console.WriteLine("Person details - {0}", person);
            person.Name = "Joe";
            person.Age = 99;
            Console.WriteLine("Person details - {0}", person);
            person.Age += 1;
            Console.WriteLine("Person details - {0}", person);
        }
}
```

We can compile the above C# example and generate the XML Documentation file:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>ConsoleApplication----XMLdoctesting</name>
    </assembly>
    <members>
        <member name="T:Person">
            <summary>
            A simple C# Class about Person</summary>
            <remarks>
            This class is created for documente testing </remarks>
        </member>
        <member name="F:Person.myName">
            <summary>
            Initialise the myName property</summary>
        </member>
        <member name="F:Person.myAge">
            <summary>
            Initialise the myAge property</summary>
        </member>
        <member name="M:Person.#ctor">
            <summary>
            The class constructor. </summary>
        </member>
        <member name="M:Person.ToString">
            <summary>
            override the method ToString() in C#</summary>
            <returns>
            which returns a string contains both Name and Age.</returns>
        </member>
        <member name="M:Person.Main">
```

```
            <summary>
            Main code for input and output</summary>
        </member>
        <member name="P:Person.Name">
            <summary>
            Name property </summary>
            <value>
            the value of Name can be set by users</value>
        </member>
        <member name="P:Person.Age">
            <summary>
            Age property </summary>
            <value>
            the value of Age can be set by users</value>
        </member>
    </members>
</doc>
```

As we can see, the XML documentation file could be very long if the source C# file is large, although it can be read, but it is not very useful in that format. Visual Studio also provides another tool to generate a Code Comment Web Report [3]. The result is a set of HTML files that can be viewed from a browser, which is simple and easy to use but contains too much information and feels tedious to read if the program is very big. The following is what the code comment web report looks like in the browser:

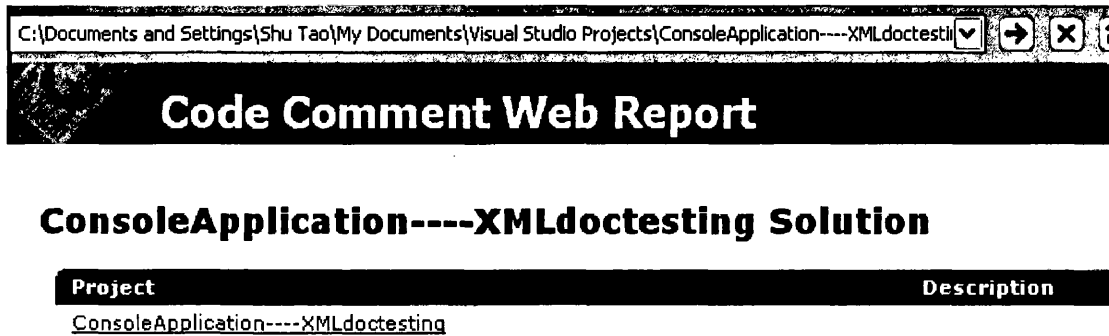


Figure1: Code Comment Web Report Main Page

After we click on the link, we will get the following details of the sample C# class in the browser. Also we can get other details of each object by clicking the links:

Person Class

A simple C# Class about Person

Access: Project

Base Classes: Object

Members	Description
myName	Initialise the myName property
myAge	Initialise the myAge property
Person	The class constructor.
Name	Name property
Age	Age property
ToString	override the method ToString() in C#
Main	Main code for input and output

Remarks:

This class is created for documente testing

Figure2: Details of the Class

Person.ToString Function

override the method ToString() in C#

Public string ToString ()

[illegible]	Description
string	which returns a string contains both Name and Age.

Figure3: Details of ToString()

This documentation provides very clear syntactic information – it completely and unambiguously describes *how* to call the methods and create an instance of the class. However, the *behaviour* of the class is not as obvious – at least, not in complex cases.

Java also offers similar API documentation [4] which specifies the syntax of the

public interface of all its built in classes together with an informal description for the semantics of the class and its methods, fields and constructors. The *javadoc* tool [77] is also able to generate API documentation in the same format for a programmer's own classes. As stated above, as with C# the syntax is formal but the informality of the semantic description means that all this automatically generated documentation can be often difficult to understand, ambiguous or even incorrect.

In order to try and solve this problem, we could extend the current documentation tools or use self-defined utilities to include formal semantics to improve the quality of the documentation. We need to note, the point of the work here is to specify and document C# programs (attributes, typing, information, etc.) but *not* the content of the programs. By that means, we can also extract a complete formal specification. If the formal specification is executable, it can also be used for testing. There are many different formal specifications that are possible, such as algebraic and equational specification, pre- and post- conditions and Z notation. Some specifications systems are executable and some are not. Also there are several tools available to reason about formal specifications such as ACL2 [5], PVS [6] and Maude. The generic tools we are trying to create are intended to be capable of generating the formal specification in many different forms, which should be highly configurable and extendible tools.

This research is based on J. Biddle's PH.D. Thesis [1], where the main interest was Algebraic Specification. Therefore, the examples in this dissertation to illustrate our tools and techniques will mostly be based on Algebraic Specification - specifically using the Maude language. However it is not necessary for our generated specification that it should be written in Maude. It is simply the format we have chosen for our examples. The process could if desired be adapted to a different form of specification - either a different Algebraic Specification language, or a different technique altogether. We will show an example based on a different specification technique in Chapter 6.

1.1 A Brief Look at Examples

The aim of the project is to produce a set of highly configurable tools that enable a range of different types of formal specification to be embedded in C# documentation. It is important that the specification notation closely adheres to the syntax and style of the programming language used in order to make it natural for a programmer to use. This we have attempted to do as much as possible with the syntax of our specification. For example, in the case of Algebraic Specification, we would like to use the following notation when

pushing an element *x* into a stack *S*:

S.push(x)

Instead of the (more usual for Algebraic Specification):

Push(S, x)

However, the second notation is commonly used in many algebraic languages. It is also important that the internal equations which define the common ancillary functions of the class (not the equations used to define the semantics of the methods themselves as written by the programmer) should be automatically generated as much as possible from the program source code.

For example, when representing the behaviour of a *public field*, we can automatically generate the required equations. Given, say, a field *i* in some class *A*, we could automatically generate something like (specific syntax being language dependent):

A.(i:=x).i = x

Where "*_._: class field -> value*" is the *field lookup operator*, and "*_.(_:=_):class field value -> class*" is the *field assignment operator*. See below for a definition of the use of "_" in these operators.

The following C# *stackOfElt* class (*an Integer Stack Class*) with the embedded specifications will be the main illustrative example through out this dissertation (note we omit the details of method implementations as these are not needed here):

```
//an example code of an int Stack Class
    public class stackOfElt
    {
        public int Elt;
        public Stack myStack;

        //constructor--create an empty stack
        public stackOfElt()
        {... }

        //constructor--create an empty stack with a element
        public stackOfElt (int Elt)
        {... }
```

```
        //return the value of the first element in the stack
        public int top()
        {... }

        //remove the first element from the stack
        public void pop()
        {... }

        //add a new element into the stack
        public void push(int Elt)
        {... }

       //return the value of the first element in the stack and remove it from the stack
        public int topAndpop ()
        {... }
}
```

Our methodology embeds specification information (ES) inside attributes (see more on Chapter 2.4.1). Here is the *stackOfElt* example with an embedded algebraic specification:

```
[Hidden("op EmptyStack: -> stackOfElt")]
public class stackOfElt
{
    [Vars("E:int; S:Stack")]
    public int Elt;
    public Stack myStack;

    public stackOfElt()
    {... }

    public stackOfElt (int Elt)
    {... }

    [Eq(("((S .push(E)) .s) .top()== E .",MType="Query")]
    public int top()
    {... }

    [Eq("((S .push(E)) .s) .pop()== S .",,MType="Command")]
    public void pop()
    {... }

    [Eq("((S .push(E)) .s) .pop()== S .",MType="Command")]
    public void push(int Elt)
```

```
        {... }

        [Eqs("((S .push(E)) .s) .topAndpop() .q == E .;((S .push(E)) .s) .topAndpop() .s ==
S .",MType="Query and Command")]
        public int topAndpop ()
        {... }

    }
```

We will use the *stackOfElt class* as a consistent example throughout this thesis. It is important to note that we are not concerned with the details of specifying stacks. In fact, specification of stacks is inherently difficult [32]. However, we are only using the *stackOfElt class* as a structural example - although the equations we use are broadly correct, they are also rather simple (to make the example clear) and not intended to handle every difficult case.

The C# descriptions of the methods themselves contain much of the information to construct specifications, such as their return types and types of parameters – that is the syntax. But that is not enough. We use the attributes (see chapter 2.4.1) in C# to embed the other necessary information and some documentation[1]. These attributes can be attached to most entities in the C# class. The following Maude code is generated by our software tools from the above C# class:

```
fmod stackOfElt is
protecting stack .
sort stackOfElt .
subsort stackOfElt < stack .
op EmptyStack : -> stackOfElt .

var E : Int .
var S : stackOfElt .

op _.top() : stackOfElt -> stackOfEltInt .

subsort stackOfEltInt < stackInt .

op _.base.top() : stackOfElt -> stackOfEltInt .
var A : stackOfElt .
eq A .base.top().q = 0 .
```

[1] We can see there are even duplicated equations existing in our original attributes (our tools are able to take out the duplications inside the ES). This is a consequence of our approach to algebraic specification (see details in Chapter 4).

```
sort stackOfEltInt .
op (_,_) : stackOfElt Int -> stackOfEltInt .
op _.q : stackOfEltInt -> Int .
op _.s : stackOfEltInt -> stackOfElt .

op _.pop() : stackOfElt -> stackOfEltVoid .

sort stackOfEltVoid .
op (_,_) : stackOfElt Void -> stackOfEltVoid .
op _.q : stackOfEltVoid -> Void .
op _.s : stackOfEltVoid -> stackOfElt .

op _.push(_) : stackOfElt Int -> stackOfEltVoid .
op _.topAndpop() : stackOfElt -> stackOfEltInt .

eq ((S .push(E)) .s) .top() .q = E .
eq ((S .push(E)) .s) .pop() .s = S .
eq ((S .push(E)) .s) .topAndpop() .q = E .
eq ((S .push(E)) .s) .topAndpop() .s = S .
endfm
```

In the above Maude example, several sorts and operations are introduced:

- That the presence of an "_" in an operation name indicates syntactically where an argument to that operation is placed;
- That the behaviour of the operations is defined by the equations – which can be conditional though there are none here; and
- That the variables are simply placeholders for the definition of the equations.

Sorts can be considered as an alternative name for type. A sort like *stackOfEltInt*, is a pair which takes two arguments (sort *stackOfElt* and sort *Int*) and combines them into a new sort consisting of a pair (see more about Maude in Chapter 2.7 and see more about this example in Chapter 4.3). We can see it could be quite a lot of work if written by hand. Furthermore we have left out large amounts of supporting structural specification necessary when, say, a class involves inheritance. This code is generated from the C# *stackOfElt* class, and it is obvious that the result of this is to successfully ease the work of writing specification and documentation.

In our illustrative examples, we try and specify C# classes by modelling the structures and some features of C#. It is our aim to provide a strong basis for specifying C# classes that in future work can be extended and improved upon. We will also attempt to model functionality that we consider interesting such as modelling C# reflections. Of course it is not possible for us to model every

feature of C# in this dissertation, but the generic framework we intended to build have the ability to model many other features if needed.

1.2 Overview of Thesis

The structure of this thesis is as follows:

Chapter 2 contains literature review and discusses the technologies that will be used. We will consider documentations and the formal specifications. We will look at the main technologies we have chosen in our project. C# is the main language used thorough this project and we will pay particular attention to the attributes and reflection. Also we will consider at XML which is used to store and transform data. Finally we will look at Maude which is an algebraic language which can be used to represent specifications.

In Chapter 3 we will have a brief look at the whole structure of our software tools. We will introduce the functions of the each part of our research, the motivation behind the design decision and present the overall scope of the work and its progress.

In Chapter 4 we will have a detailed look at the modelling system of our generic toolset. We will show how we can transform a C# class into an XML specification and then turn the XML file into another formal specification - specifically using algebra. We will first examine how we fully model the basic structure of a class. We will then examine how we fully model the inherited features of a class. We choose Maude as a target language to illustrate the modelling process.

In Chapter 5 we will look at examples C# classes that we have pre-defined. We will look at modelling arrays and how to generate equations for arrays. We will look at a simple example to model multiple levels of inheritance.

In Chapter 6 we will summarize the work and suggest areas where further work can be done to build upon the work we have presented here. We will propose a prototype of the universal object-orient class model which can be used with other formalisms. We will consider a pre/post-condition documentation/specification model.

Chapter 2 Literature Review and Overview of Chosen Technologies

This chapter discusses the main technologies we have used in this project and explains the reason we choose them. It also surveys the relevant literature.

2.1 Object-Oriented Programming and C#

In this section we will give a brief overview of the current state of object-oriented programming languages. We will pay particular attention to C# and will explain why we have chosen C# as the language that we wish to specify.

2.1.1 The Current State of Object-Oriented Programming

The first example of object-oriented programming dates back to Simula 67 in the 1960s [78]. But the first industrially successful object-oriented programming language was C++ [7] [8]. The intention when it was designed was to make sure that the language would achieve a wide acceptance. One of the key factors in this was designing it to be a superset of the language C which was and is a very popular language particularly in its use for the development of the UNIX operating system [9] [10]. This means that both C and C++ code can be compiled in a C++ compiler [10].

Another popular object-oriented language in large commercial use today is Java. Java was developed by Sun Microsystems [11]. Java is a language that supports multi-threading, distributed programming and better security than C++ provided [12]. The chief advantage of Java and the one Sun Microsystems identifies as its core feature is the "Write once, run anywhere" (WORA) [13] principle. A Java program can be written on any platform and then can be run on any platform which has a Java Virtual Machine installed provided certain rules are obeyed.

2.1.2 C#

Compared with Java and other object-oriented language, C# is the newest object-oriented programming language which is rapidly growing in popularity in recent years [14] [15]. C# is a strongly typed objected-oriented programming language written by Microsoft to target the .NET platform [16]. As a new language, C# draws on the lessons learnt on the past many years and has the influence of Java, C++ and many other languages. Microsoft expects C# combines strength of C (high performance), C++ (object-oriented structure), Java (high security), and Visual Basic (rapid development) [17]. It is intended to improve on the structure of C++ and especially Java in terms simplicity and performance. For Java programmers, the C# syntax is very similar and the semantics are familiar. One advantage C# has over Java is that it provides a better object-oriented syntax for much faster accessing member fields of an object than Java does using properties (It is certainly more convenient and syntactically more concise).

One of the important features of any object-oriented language is its support for defining and working with classes. Classes define new types, methods and constructors which allow the programmer to implement encapsulation, inheritance and polymorphism [18].

As stated in Chapter 1, C# supports a new XML style of inline documentation that simplifies the creation of online and print reference documentation for an application. C# also provides component-oriented features, such as properties, events, reflection and declarative constructs (called attributes) [18]. This feature benefits us a lot since attributes and reflection are both principle technologies we used in our project. We will explain them in detail in the following sections.

2.2 Documentation

When we build software or even just write a short program, code alone is insufficient. Also we may wish to enhance the software, either by improving present facilities, or by adding new ones. Furthermore, it may later become the responsibility of programmers who did not originally write it. Comprehensive documentation is usually considered to comprise these activities of error correction and enhancement [61]. Software documentation normally contains three parts:

- Historic Documentation: This contains information about the creation of the

software, including general strategic plans and updates.

- System and Programming Documentation: This gives full details of the program, describing various aspects of the class, fields, methods and so on. And this is the part relevant with our research. (Of course, classes, fields and methods are only relevant to Object-Oriented programming)
- Test Documentation: This is a record of tests designed to prove that the software is working correctly, which includes test data and expected results.

Documentation is usually embedded within the source code and stays in the form of comments. For example, as we saw in the first example of Chapter 1, we can use triple slash (///) to write XML documentation (see Chapter 1). Most programmers would write this documentation while producing the source code. There are quite a lot of programmer software tools providing the utilities to integrate the documentation and source codes, and also able to automatically generate the documentation for programs, such as specifications of functions and parameters. Examples are the Code Comment Web Report for C# and javadoc for Java (both see Chapter 1).

It is important for a good program to have a clear, simple and detailed documentation which is not tedious to read, it should be able to answer all questions about the operational performance of the software. Otherwise it will be hard to maintain the source code once we want to update it. Also professional documentation should define and explain the application programming interface (API), data structures and even the algorithms. The documentation should be *precise and unambiguous.* That is a key reason for including formal, mathematical documentation.

Literate programming is a well known concept of software documentation, which is introduced by Donal Knuth in 1981 [62]. Literate Programming tries to combine the programming source code and the documentation together, which makes the program primarily for humans to read easily. This contrasts the traditional idea that we create the source code for computers to read [64] [65]. The following example is a simple program to calculate the area of a circle:

- clearscreen

- print text Please type your name:

- store input

- print Hello there,

- print value

- print .   Nice to meet you.

- newline

- newline

- print text Let's work out the area of a circle.

- newline

- print text Please enter the radius of the circle in furlongs:

- store input

- multiplyby value

- multiplyby 3.14159

- print Thank you

- newline

- print text The area of the circle is

- print value

- print text square furlongs.

- newline

- newline

This example is a basic literate programming, which runs in the query-response mode of operation. First, it asks the name of the user and gives greetings, and then it asks the value of radius of the circle and calculates the output. As we can see from this example, the basic principle of literate programming is easy to be read by human from the program itself. However, for complex cases, it is not clear that this form of documentation is sufficiently precise, or even that easy to read.

2.3 Formal Specifications

A formal specification is a description of a program in a mathematical form. It should precisely define some aspect of a system: typically, what the system is intended to functionally do, though other aspects could be defined (for example, in the case of concurrent/parallel systems, *safety* and/or *liveness* properties). Formal specifications can be used to prove the correctness of a system by formally showing that the specification and the corresponding implementation have the same behaviour [68].

There are several formal specification methods in existence. For example:

- Pre- and Post- Condition
- Algebraic and Equational Specification
- Z Notation
- B Method
- Process Calculus

Design by contract (DBC) is a methodology for designing programs on the basis of mutual *obligations and benefits* based on Pre- and Post- Conditions [45] [46] [47]. By applying this principle to an object-oriented program, there is a client

and a supplier who both agree on a contract. The supplier has to satisfy some obligations for the client and the client has to give the corresponding benefits. The functions and procedures specify the Pre-condition and Post-condition. The Pre-condition is an obligation for the client and a benefit for the supplier. The Post-condition is an obligation for the supplier and a benefit for the client. This concept is defined by Bertrand Meyer with his design for the Eiffel programming language [48]. DBC has been accepted as a common concept of object-oriented languages and can be used in most popular object-oriented languages such as Java, C++ and C#. All the classes if design by contract is used, would define these relationships between Client classes and Supplier classes. A Client class can make calls to Supplier classes which can return the state of the Supplier class. And the Supplier class can provide a return state and data that is required by the Client class. Pre- and Post- Conditions can be used as a formal specification to verify if the implemented code does precisely do what the requirements claimed (see more on Chapter 6). The Java Modelling Language (JML) [81] which also embeds pre/post conditions – as well as other stuff – in javadoc comments.

Algebraic specification is an alternative formal method, and the one that will be principally used in this thesis. Let us have a look at *a single-sorted algebra* first [1, 49, 50, 51, 52, 53, 54, 55]:

B = (B | true, false, not, and)

We take the algebra *B* of Booleans with sort *B = {true, false}*, constants *true* and *false*, the negation function *not* and the function *and* defined as follows:

not(true) = false
not(false) = true
and(true, true) = true
and(true, false) = false

and(false, true) = false

and(false, false) = false

And here is a similar *many-sorted algebra*:

Nat Bool = (N, B| true, false, not, and, succ,+, eq)

eq: $N^2 * B \rightarrow N$

$$eq(a,b,c) = \begin{cases} a & if(c = true) \\ b & if(c = false) \end{cases}$$

Algebraic specification is used as the basis of a range of tools and languages. The Common Algebraic Specification Language (CASL) is an algebraic specification based on first-order logic [66].

CASL was designed by CoFi (The Common Framework Initiative) [66] by introducing Partial functions and subsorting. It is an expressive language and can be used to specify conventional software. The following example is the stack class in CASL [58]:

```
spec
   Stack[sort Elem] =
   sort
      stack
   ops
      empty: stack
      push: stack * Elem -> stack
      pop: stack ->? stack
      top: stack ->? Elem
   axioms

      not def pop(empty)
      not def top(empty)
      forall s : stack; e : Elem
          · pop(push(s,e)) = s
          · top(push(s,e)) = e
end
```

The above specification is for a stack example of sort *stack* and with four operations:

- Operation *Empty*: create an empty stack
- Operation *Push*: add a new element into the stack
- Operation *Pop*: a partial function remove the top element
- Operation *Top*: a partial function return the top element

A partial function is the one which is not defined on all its arguments. For example *"squareroot:N->R (N=naturals, R=reals)"* is a total function, because all natural numbers (*i.e. >= 0*) have square roots but *"squareroot:Z -> R (Z=integers)"* is not because negative numbers do not have square roots.

When we define the axioms, for partial functions *top* and *pop*, we specify them to say they are not defined on empty stacks. Also we define the equations for *pop* and *top*. The principle design paradigm is to use Abstract Data Types (ADT) to specify the function and design of software packages. CASL consists of three

types of specification [1] [57] [66]:

- Basic Specification: specifications for single software modules
- Structured Specification: present the specifications in modular style
- Architectural Specification: allow a large specification to be represented in small logically organised smaller specifications.

Finally each of these three groups of specifications can be gathered together into libraries allowing the storage and distribution of named specifications [63].

Another important Algebraic specification language is Maude (see more in Chapter 2.7) which we have already introduced previously. Maude is a high level language that provides support for writing specifications that can then be executed using rewriting logic [40]. Maude also provides support for equational specification as Maude uses membership equational logic. Maude use functional modules to define theories in equational logic using equalities of the form $t = t'$ and conditional equalities of the form $t = t'$ *if* t'' ; and also membership axioms of the form *t:S* where t is a term of sort S (there are also conditional membership axioms). Maude can be used to module object-oriented systems by using its functional modules.

Consider the sorts *Z* of integer and *N* of natural numbers. We can define a *subsort relation* between these: "*N < Z*" because all natural numbers are also integers - though not the other way around. A conditional membership axiom might be: "*a:N if a >= 0*", where a is a variable of type *Z*. That is, we can *narrow* a variable of one sort (Z) to another more restricted sort (N) if a condition is met (*a >=0*).

Another important specification formalism is the Z notation. The Z notation is a formal specification language used for describing and modelling computing systems [67]. Z was originally proposed by Jean Raymond Abrial in 1977 [59] and has been developed at the PRG (Programming Research Group) at the Oxford University Computing laboratory (OUCL) afterwards. Z is based on the standard mathematical notation used in axiomatic set theory, lambda calculus, and first order logic [60]. However, Z is a non-executable specification language.

The B method is a formal method based on AMN (Abstract Machine Notation), which supports development of programming language code from formal specifications [67] [70] [71]. The B method is similar to the Z notation since they were both developed by Jean Raymond Abrial. But the B method is more focused on the programming code rather than on the formal specification. There

is a good tool support for the B method to make it easier to implement a formal specification, which makes B practically useful in commercial design, such as in the Paris Metro Line 14 [79].

The last formal method we would like to talk is the family of the Process calculus. In computer science, Process calculus is mainly related to formally modelling concurrent systems, which includes π calculus, CSP and CCS. CSP (Communicating Sequential Processes) is a formal language for describing patterns of interaction in concurrent systems [72] [73]. CSP was developed by C. A. R. Hoare and has been practically used in industry as a tool for specifying and verifying concurrent systems. CCS (Calculus of Communicating System) was developed by Robin Milner, which models indivisible communications between exactly two participants [74]. Later on, Robin Milner, Joachim Parrow and David Walker together developed π calculus based on the work of CCS. π calculus is able to model concurrent computations whose configuration may change during the computation [75]. π calculus could be a universal model of computation and has been applied into several practical applications such as modelling business processes (which is also known as the basis of Business Process Modelling Language).

Formal specifications have been proved to be a practical way to modelling and testing specific systems when applied to certain kinds of problem: mainly *safety critical applications,* where the need for specific mathematical expertise is outweighed by the consequences of system failure. What is lacking at the moment is there are no industrial toolsets to make development using formal specifications faster and easier. It is hard for people to understand and use specifications if they do not know any specification language. At the moment it is generally necessary to have substantial mathematical knowledge of the underlying logics used by the languages. This project is trying to make a generic toolset which is possible to make the task of producing formal specifications in a reasonable and easier way.

2.4 Attributes and Reflection in C#

This project relies quite heavily on the *attribute* and *reflection* technologies of C#. In this section, we discuss and explain these technologies.

Before discussing attributes and reflection, we need to introduce Metadata. Metadata is information about the data stored along with your program, which includes data about the types, code, assembly and so on. Attributes are a mechanism for adding metadata, such as compiler instructions and other data about methods, and classes. Reflection is the process by which a program can

read its own metadata, which is mainly about extracting metadata from a program itself.

2.4.1 Attributes

An attribute is an object in a C# program, which represents data that you want to associate with some elements or information of the code. The elements are referred to as the target of that attribute. For example, the pre-defined attribute *[WebMethod]*, which is used in C# Web Service programs to invoke a method as a web service that is able to provide proper functions in the web browser [19]:

```
[WebMethod]
Public string HelloWorld()
{
    Return "Hello World";
}
```

Attributes come in two forms: pre-defined and custom. Pre-defined attributes are supplied as part of the Common Language Runtime (CLR), and they are integrated into C# working as keyword-like descriptive declarations. So *[WebMethod]* is a pre-defined attribute [17] [20] [21]. Most C# Programmer would normally just use these pre-defined attributes to modify the C# source code. Custom attributes are attributes we create for our own purposes.

Custom Attributes are very powerful tool in C# and here we used it as one of the main technologies in this research. For example, in a small library, we have a database to record all the books, but we may have some bugs in these records and we want to link these bugs report to specific fixes in the code. Then we can do like [17]:

```
[FixAttri(4978, "Computer Graphic v2", "2/21/2006", "John Lee"
Comment = "Book Writer Edited"]
```

We can now write a program to read through the metadata to find these attributes and update the database. The attribute would serve the purposes of a comment, but also allow us to retrieve the information programmatically through reflection which we will discuss later.

Like most things in C#, attributes are embedded in the classes. Before we create a custom attribute in our program, we need to derive the new custom attribute class from *System.Attribute*, for example [22]:

```
public class FixAttri : System.Attribute
```

We also need to tell the compiler which kinds of elements this attribute can be used with by specifying the attribute target [23]:

```
[AttributeUsage(AttributeTargets.Class |
        AttributeTargets.Constructor |
        AttributeTargets.Field |
        AttributeTargets.Method |
        AttributeTargets.Property,
        AllowMultiple = true)]
```

In this example, *AllowMultiple* is set *to true*, indicating that class members can have more than one *FixAttri* assigned. In this case, *FixAttri* can be attached to classes, constructors, fields, methods and properties. Note that *AttributeUsage* is itself an attribute.

Every attribute must have at least one constructor. Attributes have two types of parameters: *positional* and *named*. In the *FixAttri* example, the comment is a *named parameter* (that is, identified as a name-value pair), and the others are all *positional parameters* (that is, identified by their position in the argument list). Positional parameters are passed in through the constructor and must be declared by the proper order in the constructor:

```
public FixAttri (int bookID, string bookName, string bookDate,
string bookWriter)
{
      This.bookID = bookID;
      This.bookName = bookName;
      This.bookDate = bookDate;
      This.bookWriter = bookWriter;
}
```

Named parameters are implemented as properties [80] and we only have Comment as the Named parameter in *FixAttri* example:

```
public string Comment
{
    get
    {
        return comment;
    }
    set
```

```
        {
            comment = value;
        }
}
```

Positional parameters are implemented as read-only properties:

```
public int BookID
{
    get
    {
        return bookID;
    }
}
```

The two types of the parameters make attributes more flexible. When using the attributes, the positional parameters must appear but the named parameters as optional.

The information in the Attributes can be considered as Embeded Specifications (ES) which we have introduced in Chapter 1.

2.4.2 Reflection

In order to access the attributes in the programs, we need to use reflection. Along with the *Type* and *TypedReference* classes in C#, reflection can dynamically create an instance of a type which binds an existing object, provides the access for examining and interacting with its metadata [24] [25].

Reflection is generally used for the following four tasks [24] [26]:

- Viewing metadata
- Performing type discovery
- Invoking properties and methods on objects which can be identified only at runtime
- Creating types at runtime

The full details of the usage of the reflection in C# are complex. In this project, we mainly use it to view metadata and discover types. Here we would give a brief idea how we would use it in C# and will describe it in more depth in the later chapters at the relevant points. Let us have a look at the following example:

```
System.Reflection.MemberInfo inf =typeof(myExampleClass);
```

The object *System.Reflection.MemberInfo* is pre-defined in C# to discover the attributes of a member and to provide access to the metadata [27] [28]. The operator *typeof* returns an object of type *Type*. The *Type* class is the root of the reflection classes and is the primary way to access metadata. Type derives from *MemberInfo* and encapsulates information about the members of a class.

```
object[] myAttributes;
myAttributes = inf.GetCustomAttributes(typeof(FixAttri),false);
```

On the above example code, we could call *GetCustomAttributes* on the *MemberInfo* object, passing in the type of the attribute we defined before in the previous examples [29] [26]. We get back an array of objects in *FixAttri*, which are *bookID, bookName, bookDate* and *bookwriter* in the proper order, for example:

```
Foreach (object attribute in myAttributes)
{
FixAttri fa = (FixAttri) attribute;

mybookID =fa.bookID;
mybookName =fa.bookName;
mybookDate =fa.bookDate;
mybookwriter =fa.bookwriter;
}
```

2.5 XML

XML is currently the main standard industry format to share and store data [30]. XML is derived from SGML (Standard Generalized Markup Language). SGML is a metalanguage to define markup languages for documents, which was developed in the 1960s by IBM [69]. SGML provides a variety of markup syntaxes that can be used for many applications, but its complexity prevented itself to be widely accepted. XML simplifies SGML (for example, by requiring all tags to be closed) while not significantly losing expressibility. In this project, we use XML files as the intermediate format to store, transfer and exchange data.

2.5.1 Easy Data Transfer and Exchange

Programs could transfer and exchange data easily if all the data are in the same format, however, most programs have historically had their own specified data format. We always need conversion programs to let applications transfer data between themselves. The data formats have become so complex and even an upgraded software can not read data from an earlier version of the same software.

In XML, data and markup is stored as a simple text file that we can access and configure at anytime. And compared with many other data formats, like Microsoft Excel which can use up to five times as much space (you may try a simple experiment by just storing some texts in Microsoft Word and a XML file), XML files are smaller and more efficient in storage. Also at the moment, many modern programming languages (for example, Java and C#) have extensive APIs for parsing, manipulating and writing XML formatted data.

2.5.2 Customizing Markup Languages

```
<?xml version="1.0"?>
<Class>
    <Name>Shape</Name>
    <Hidden>
        <Operation>op AShape : -> Shape</Operation>
        <Comment />
    </Hidden>
</Class>
```

The XML above is a simple example of the notation that will be used in this thesis. As can be seen, the Markup is customizable. We have introduced the *Class*, *Name, Hidden, Operation* and *Comment* tags – these do not pre-exist in XML. Note also the syntax of the empty Comment tag. So, if someone creates a data file based on XML, we can add the extension whatever we want easily, and which shows XML is very flexible [33]. For example, we can change the tag *<Hidden>* to *<HiddenOperation>* and extend the example with new tag content *<MethodName>*:

```
<?xml version="1.0"?>
<Class>
    <Name>Shape</Name>
    <HiddenOperation>
        <Operation>op AShape : -> Shape</Operation>
```

```
            <Comment />
        </HiddenOperation>
        <MethodName>Area</MethodName>
</Class>
```

2.5.3 Self-Documenting Data

The data in XML files is *self-describing* if we pick good (semantically appropriate) tag names. Let us have a look at the following example which takes from the project:

```
<?xml version="1.0"?>
<Class>
        <Name>stackOfElt</Name>
        <Hidden>
                <Operation>op EmptyStack -&gt; StackOfElt</Operation>
                <Comment />
        </Hidden>
        <Field Type="int">Elt</Field>
        <Field Type="Stack"> myStack</Field>
        <Method Type="Query">
                <Name>top</Name>
                <ParameterType />
                <MethodReturnType>System.Int32</MethodReturnType>
                <Comment>Return the top element of the stack</Comment>
        </Method>
</Class>
```

<?xml version="1.0">, is standard XML processing instruction which indicates that this is a XML file and the version is 1.0. At the moment, this is the only version available. Based solely on the names of each XML element in the above example, we can determine fairly easily what is being represented. This example is simply given some basic information about a C# class, like <Name>stackOfElt</Name> tells us the class name is *stackOfElt*. Note that although XML files can be quite descriptive if good tag names are chosen, they are not that easy for humans to read. (For notation ">" see Chapter 2.5.5)

2.5.4 Structured and integrated Data

Another useful aspect of XML is that it lets people to specify not only data, but also the structure of data and how various elements are integrated into other elements. This is very useful and important when we are dealing with complex data.

```
<?xml version="1.0"?>
<StudentList>
    <UpdateDate>02192006</UpdateDate>
    <Student>
        <Number>101</Number>
        <Name>James</Name>
    </Student>
    <Student>
        <Number>103</Number>
        <Name>Linon</Name>
    </Student>
</StudentList>
```

In the above example, each *<Student>* element needs to enclose a *<Number>* and a *<Name>* element. But the *<UpdateDate>* element can not move into the *<Student>* element. This emphasis on the correctness of documents is strong in XML and we can use DTD (Document Type Definition) or XML schemas to make sure the data in XML file is kept in the proper format [33]. Most XML browsers are able to check the XML files to see if they are well-formed or valid according to a particular DTD or schema.

2.5.5 The Five Pre-existing Entity References

There are five pre-defined *entity references* in XML. An entity references is replaced by the corresponding entity when the XML document is processed. We would like to note them here as they appeared in our XML Class Specifications. The following are the five pre-defined entity references in XML and the corresponding characters they are replaced:

- < replaces <
- > replaces >
- & replaces &

- ' replaces '
- " replaces "

2.6 XSL

The Extensible Stylesheet Language (XSL) is a language for defining and transforming XML document [35]. XSL has two abilities: transformation and formatting [35]. The transformation part is able to transform the structure of XML documents into different forms such as PDF and HTML, and the formatting part is able to format and style the XML document in various ways [36]. In this project, we simply use XSL to manipulate our XML documents and transform them into a formatted file that is easy to read. For example, in some popular browsers like IE and Firefox, the XML documents we generate would be simply displayed using the XML as we created it:

```
<Hidden>
        <Operation>op AShape : -> Shape</Operation>
        <Comment />
</Hidden>
```

But we can tell the browsers how to display the elements we have created in the XML document by using XSL. We need to add *<?xml-stylesheet?>* process instruction to indicate the XML ducoment is using XSL.

```
<?xml-stylesheet type="text/xsl" href="classSpec.xsl"?>
```

Also we need to set the type attribute and the href attribute which defines the uniform resource identifier (URI) of the XSL file. The following is the main code in the XSL stylesheet file for the hidden example:

```
<xsl:for-each select ="Hidden">
        <P>
        <B>Hidden </B>{
        </P>
        <P>
        <xsl:apply-templates/>
        </P>
        <P>
        }
        </P>
        </xsl:for-each>
```

```
<xsl:template match="*/Operation">
    <P>
    <xsl:value-of select="."/>
    </P>
    </xsl:template>

<xsl:template match="*/Comment">
```

Since in this project, we only use XSL as a tool to make some simply though important, for readability, visual change of our XML documents, it is not necessary to describe it in depth. We would have the following example displayed in a browser after we apply the XSL stylesheet to the Hidden code:

```
Hidden {
op AShape : -> Shape
//
}
```

Note that here the XSL transforms the code into HTML, and that the example above is showing a representation of the formatted HTML – not the source code.

2.7 Maude

As we have already stated, there are many formal languages available for formal specification, Maude is just one of them. We have chosen Maude to specify classes algebraically in our examples in order for them to fit with the long history of algebraic research at Swansea University [1] [37] [38] [39]. Also Maude allows us to write specifications we can execute for testing and analysis purpose.

Maude is an executable language supporting both equational specification and term rewriting systems for a wide range of applications [40], though here we only use the equational specification capabilities. A main aspect of this project is researching the embedding of documentation and specification within programs, and Maude can be regarded as an equational logic sublanguage which just fits the needs of this research when we consider, as a major example, the embedding of algebraic specifications within programs. Note that there are many features of Maude we do not use. The following is a simple Maude module for the Natural numbers [41]:

```
fmod BASIC-NAT is
        sort Nat .
```

```
    op 0 : -> Nat .
    op s : Nat -> Nat .
    op _+_ : Nat Nat -> Nat .

    vars N M : Nat .

    eq 0 + N = N .
    eq s(M) + N = s(M + N) .
endfm
```

A basic Maude functional module starts with "*fmod name is*" and ends with "*endfm*". The example *BASIC-NAT* introduces a new sort called *Nat* and three operators. The first operator "*op 0 : -> Nat .*" is a function without arguments, and is therefore a constant since it always returns the same value 0. The second operator "*op s : Nat -> Nat .*" is intended to represent the successor function, for example, we can define constants "*op 2 :-> Nat .*" which can later be defined by an equation "*eq 2 = s(s(0)) .*". The third operator "*op _+_ : Nat Nat -> Nat .*" is intended to represent the addition function, it can be considered as + takes two arguments: "*+(_,_)*". The "_" characters mark the position of the arguments. So we would write equations like "*s(0) + s(0) = s(0+s(0)) .*". Although we can infer the meaning of the functions from their names, they are actually defined by the pair of equations.

Maude supports an extensible algebra of module composition operations, which can be used to create executable environments for different logic models and theorem proving [40] [41]. The features of Maude lay a heavy emphasis on advanced meta-programming and meta-language applications [40] – features we will not use here. We are interested in using Maude to *model behaviour,* which can be done adequately by using functional modules. Mapping from the C# Class with ES to a Maude Specification will involve lots of code and modelling work. We would wish to make as much of that as automated as possible since we intend to make the most robust generic tools in the end. Also we wish to minimize the work of the programmer in writing an embedded specification.

Chapter 3 The Overview of the Research

In this chapter we would like to explain the overall concepts of this project and illustrate it with a simple case study. In addition, we will give a brief overview of each utility we have created so far and introduce the complete structure of the current state of the project.

3.1 Motivation

The initial motivation for this project was to build a well-engineered application based on the algebraic models in [1]. The actual application has become more generic in the project progressed (see Chapter 4). We need to note, the point of the work here is to *document and specify* C# programs (attributes, typing, information, etc.) but *not* to extract behaviour from the content of the programs (which would basically be equivalent to the Halting Problem and hence not possible). Furthermore, the intention is not complete verification of C# programs. By embedding specification and documentation information in C# programs, we can minimize the effort required in building specifications and documentation by taking advantage of the syntactice information already present in the program – which we would otherwise have to repeat. We will use the original algebraic example as an illustrative study.

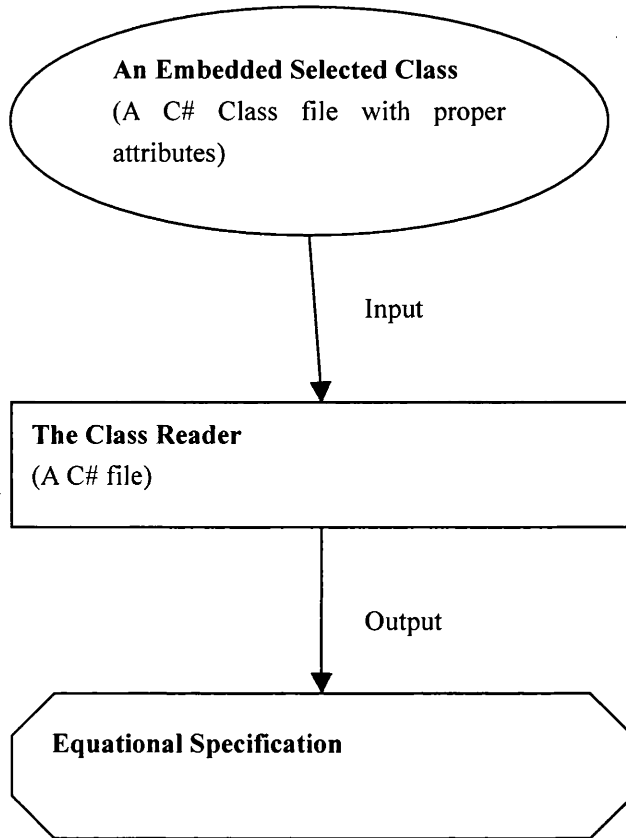
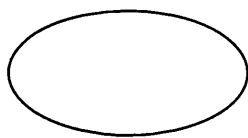
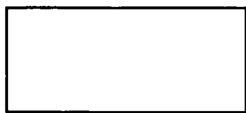


Figure3_1: Stage 1 of the System

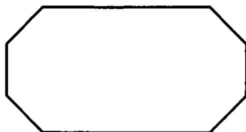
The Figure3_1 shows the basic structure of the planned application, explained it in detail below. Noted that in all the diagrams of this dissertation, we use the following shape conventions:



: Ellipse represents a configurable file in our system.



: Square represents a software utility we created to do a specific job, which normally is not configurable and functioning as an executable tool.



: Octagon represents an output which is normally an automatically generated specification or file.

An Embedded Selected Class (ESC)

As already described, we use C# attributes to embed the specification information in a C# class, which we denote the *Embedded Selected Class* (ESC). In the end product, the user should able to select any C# class and embed it with the proper attributes defining a formal specification which have been pre-defined, either by themselves or (more probably) some other programmer. In this chapter, we will use a *stackOfElt* class as the example. The following example code has two methods (*top* and *pop*) from the *stackOfElt* class:

```
[Eq("((S .push(E)) .s) .top()== E .",Comment="S is myStack, E is Element.")]
        public int top()
        {... }

[Eq("((S .push(E)) .s) .pop()== S .",Comment="S is myStack, E is Element.")]
        public void pop()
        {... }
```

In this example, the two methods are each tagged with an *equation*, defining the formal specification and documentation (actually, only *part* of the formal specification and documentation in this case) as well as a free text comment. Notice that both equations refer to another method (*push*) which is not present above.

Our application must extract the information in the attributes, together with other information already *(implicitly)* present in the program code to construct an equational specification. This is achieved by the *Class Reader.*

The Class Reader

The Class Reader is the first utility we have created in our project. It simply extracts the information we want from the Embedded Selected Class and puts it into a file.

Equational Specification

Generated by the Class Reader, this is a text file containing all the data (equations in this case) embedded in the attributes we have introduced. For example, we would have the following equations generated from the information encoded by the attributes of the above two methods (top and pop):

```
((S .push(E)) .s) .top()== E .
((S .push(E)) .s) .pop()== S .
```

At this stage, the Equational Specification file is very simple. It is necessary to add other information from the program code by using other applications in the later stages.

3.2 Bringing in the XML Specification

The output of the Class Reader is only a list of equations. We need a well-defined formal format to represent the C# class we have selected in a widely useful way. Since there are many merits of XML that we have already introduced on Chapter 2, an XML specification would be the best choice for us because it is easy for us to convert XML into other formats later on. At this stage, we refine the intended behaviour of the Class Reader.

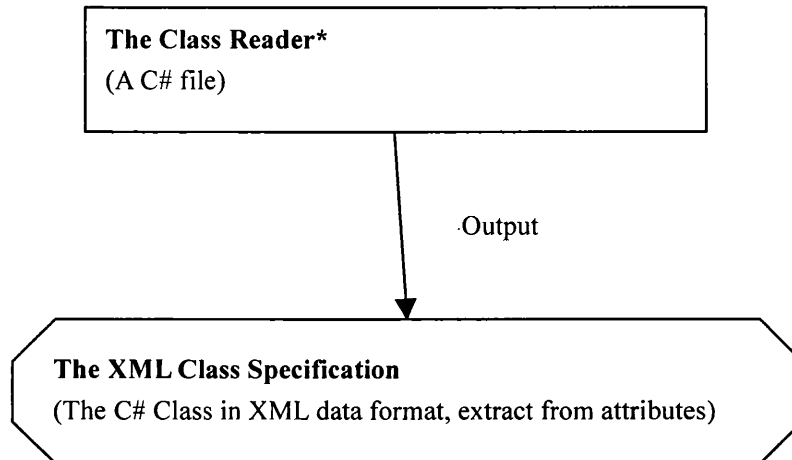


Figure3_2: Stage 2 of the System

Now the Class Reader generates XML-formatted data (the ESC is still the same so we did not include it in Figure3_2). The main progress we have made at this stage is in extracting embedded modelling information from the C# class into an XML representation of the class specification. We will explain what changes we have made in detail below.

The Class Reader*

The proposed functionality of the Class Reader at this stage has been greatly expanded. It has all the definitions of all the attributes we will need. It also has the required functions to extract the information for the Embedded Selected Class and output it into the format of the XML Class Specification. The information to be extracted is both embedded in user-defined attributes, and also - using reflection - from the actual program itself (for example, method names, parameters etc.).

The XML Class Specification

Generated by the Class Reader, an *XML Class Specification* is an XML file containing all the data and equations extracted from the class and its attributes.

Now we have already extracted modelling information from both embedded attributes and also the actual program source text of a C# class into an XML data format, let us consider the example of two methods (*top* and *pop*) in the XML class specification:

```
<Method>
    <Name>top</Name>
        <ParameterType />
        <MethodReturnType>System.Int32</MethodReturnType>
        <Equation>
                <LHS>((S .push(E)) .s) .top()</LHS>
                <RHS> E .</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>S is myStack, E is Element.</Comment>
</Method>
<Method>
        <Name>pop</Name>
        <ParameterType />
        <MethodReturnType>System.Void</MethodReturnType>
        <Equation>
                <LHS>((S .push(E)) .s) .pop()</LHS>
                <RHS> S .</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>S is myStack, E is Element.</Comment>
</Method>
```

In this example, the two methods are defined in XML format by the sequences of Name, typed of the parameters, return types of the methods, equations (include the condition of the equations) and the comments.

Note that it would be desirable to construct an XML Schema defining the contents of the XML file. We omit this here, as it is just for illustration. However, in Chapter 4, we will define a schema.

It is not easy to check the result by reading this XML file if the original C# class is very big (XML is general hard for humans to read). Here we could use an XSL stylesheet to transform the XML into a more readable form. The following is the result when we use the stylesheet (see more about XML stylesheets in Chapter 2.6 and Chapter 4.2.2) into a "generic" (that is not in any specific language) algebraic specification:

Method top : -> System.Int32{

```
eq ((S.push(E)).s).top() = E.
IF (NULL)
//S is myStack, E is Element.
}

Method pop : -> System.Void {
eq ((S.push(E)).s).pop() = S.
IF (NULL)
//S is myStack, E is Element.
}
```

This result with the XSL Stylesheet is obviously more readable, as we can easily check the types, return-types, parameters and functions the two methods have. Furthermore, we can use other XSL Stylesheets to translate the XML into a wide range of other languages (for example, Maude).

At this stage, we have embedded specification/documentation information by using attributes (which we have not yet defined) and the actual program source, and partially processed the results, generating a simple algebraic specification. By choosing a different set of attributes, and a different XSL mapping, we could produce other forms of formal specification/documentation.

3.3 The Executable Algebraic Specification

The main aim of this thesis is to investigate the process of building a generic software framework which can provide formal specification and documentation in the form that users want for object-oriented classes. In Figure3_2, we have modelled the ESC into the XML class specification, and then the XSL stylesheet converted the XML Class Specification into a "generic" Algebraic Specification, but we still want an *executable* specification which can help us in testing and analysis. Note that although we consider executable specifications desirable, not all researchers agree, for example the arguments are [82] [83]:

- Executability may limit the expressive power of a specification language and restricts the forms of specifications that can be used.
- Though executable specifications permit early validation with respect to the requirements by executing individual test cases, proving general properties about a specification is much more powerful.
- Executable specifications can unnecessarily constrain the choice of possible

implementations.

- It could be easier to verify that an implementation meets a more abstract specification, than to match an executable specification against an implementation for which possibly different data and program structures have been chosen.

In choosing an executable algebraic specification language there are many possibilities but we have chosen the executable language Maude (see more about this in Chapter 4.3).

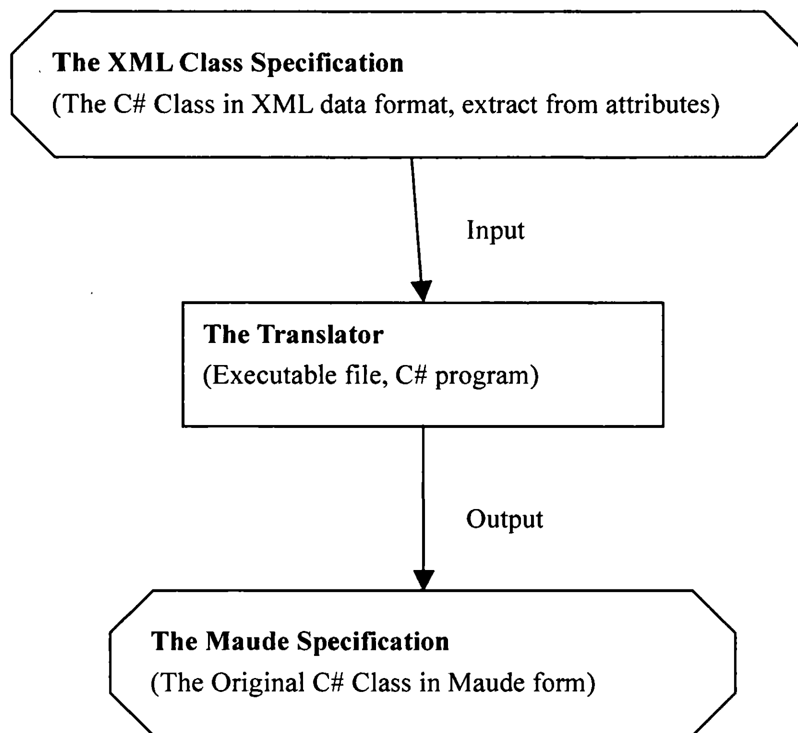


Figure3_3: Stage 3 of the System

At this stage, we will use the XML class specification as an intermediate representation. We have created a new tool called *the Translator* which can convert the XML class specification into another formal specification language – i.e. in this case Maude.

The Translator

The translator is a standalone utility we created to output a Generated Formal Specification (GFS). The Translator helps us to automatically model the XML Class Specification into Maude Specification. At this stage, we have already started to try to automate as much as possible the generation of the

specifications and documentations. Note that it is not possible to simply use XSL to generate the Maude code (as with the "generic" algebraic specification notation) because we intend to substantially expand the detail and sophistication of the algebraic representation in Maude.

The Maude Specification

The Maude Specification is generated by the Translator from the original C# class in stages. The following is a fragment of the Maude code generated:

```
op _.top() : stackOfElt -> stackOfEltInt .

sort stackOfEltInt .
op (_,_) : stackOfElt Int -> stackOfEltInt .
op _.q : stackOfEltInt -> Int .
op _.s : stackOfEltInt -> stackOfElt .

op _.pop() : stackOfElt -> stackOfEltVoid .

sort stackOfEltVoid .
op (_,_) : stackOfElt Void -> stackOfEltVoid .
op _.q : stackOfEltVoid -> Void .
op _.s : stackOfEltVoid -> stackOfElt .

eq ((S .push(E)).s).top().q = E .
eq ((S .push(E)).s).pop().s = S .
```

Noted that *stackOfEltInt* represents a *tuple type* necessary to model methods that both modify an object's state and return a value (for more details see Chapter 4.3.5).

We have so far been able to model a C# class with ES using XML and an Algebraic Specification language. At every stage, the software framework we intended to build is more integrated and generic. But we still want to make it more configurable to fit the needs of other formal specification.

3.4 A More Generic Approach

We have made our XML Class Specification and Maude Specification (the GFS) both automatically generated. We let the users choose the original C# class and embed the attributes which we define. Obviously, there is currently a restriction in that the attributes are currently "hard coded" for algebraic specification. C#

gives us the ability to define our own attributes. Thus we should able to define different attributes and hence use different specification techniques. If we do so, our software tools become a highly configurable utility – able to handle a range of formal specification techniques.

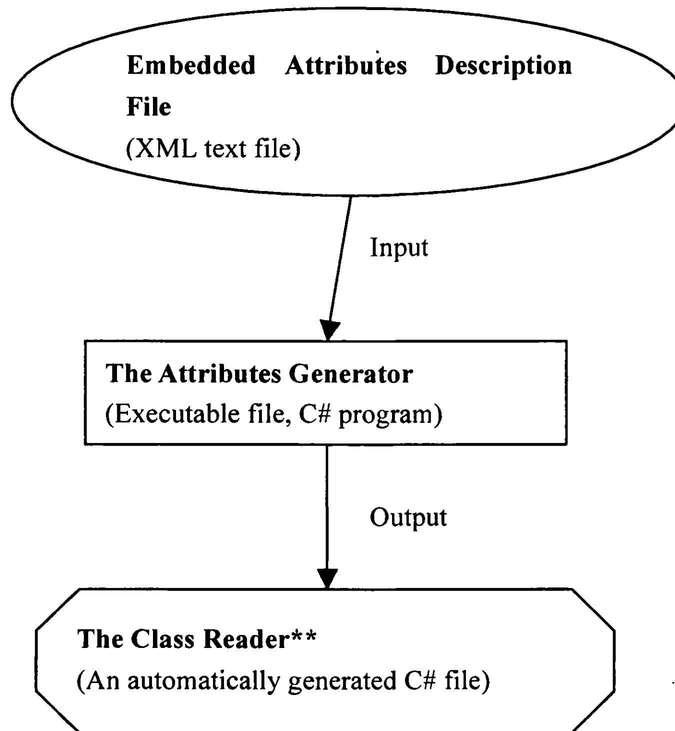


Figure3_4: Stage 4 of the System

Embedded Attributes Description File (EADF)

To extend the generic capability of our system, we add a further XML file. The *Embedded Attributes Description File* (EADF) is an XML file, configurable by users, which contains full details of data of attributes and other necessary information. The EADF works as a configuration file for the *Attributes Generator* which will create a specific set of attributes that can be attached to a C# class. The intention here is to allow users to more easily to define their own attributes and hence the specification system they are interested in. The following is an example which defines two attributes and their parameters (see more details including the XML schema of EADF in Chapter 4.1.1):

```
<Attribute>
```

```
        <Name>Hidden</Name>
        <Compact Type="accessor">operation</Compact>
        <Compact Type="parameter">comment</Compact>
        <Constructer />
    </Attribute>
    <Attribute>
        <Name>MethodComment</Name>
        <Compact Type="Parameter">mtype</Compact>
        <Compact Type="parameter">comment</Compact>
        <Constructer />
    </Attribute>
```

The two attributes defined are *Hidden* and *MethodComment*. They are simple attributes and only contain 2 properties (the properties have 2 types, see more in chapter 2.4.1 about attributes).

The Attributes Generator

The Attributes Generator is also a standalone utility that reads data from an EADF description file and generates all the attributes in C# codes for the Class Reader. That is it generates the C# source code of the Class Reader with the same functionality as the one we have written by hand earlier. The user can then embed these (generated) attributes within their own C# code.

The Updated Class Reader[2]**

The Class Reader here is ideally to be able to be fully generated by the Attributes Generator, but also has the functions to extract the information for the Embedded Selected Class and output them into the format of the XML Class Specification. The following is the code automatically generated (for the detailed explanation of the following code see Chapter 4.2):

```
public class Hidden: System.Attribute
{
string operation;
string comment;

public Hidden
(string operation)
{
this.operation = operation
```

[2] The Class Reader considered here has the same functional ability as the one in Chapter 3.2. The difference between them is the Class Reader here is automatically generated by the Attributes Generator, and the previous one is written by hand. Therefore, we can see the code is not well formatted and without comments since it is not generated for human reading. In addition, the member names (e.g. Aoperation) are automatically generated also, and hence not chosen to be easy for humans to read.

```
}

public string Aoperation
{
get
{
return operation
}
}

public string Acomment
{
get
{
return comment
}
set
{
comment = value;
}
}
}

public class MethodComment: System.Attribute
{
string mtype;
string comment;

public MethodComment
()
{
}

public string Amtype
{
get
{
return mtype
}
set
{
mtype = value;
}
}
```

```
public string Acomment
{
get
{
return comment
}
set
{
comment = value;
}
}
}
```

3.5 Overview of the Work in this Thesis

The following diagram is the structure of the current stage of our project. As we can see, the project still has a lot of space to expand.

The Attributes Generator reads the EADF to generate the Class Reader. The Class Reader reads the ESC to output the XML Class Specification file. We may use XML stylesheets to generate the "Generic" Algorithms Specification. In order to produce an executable specification, the translator is created to model the XML Class Specification into the Maude Specification.

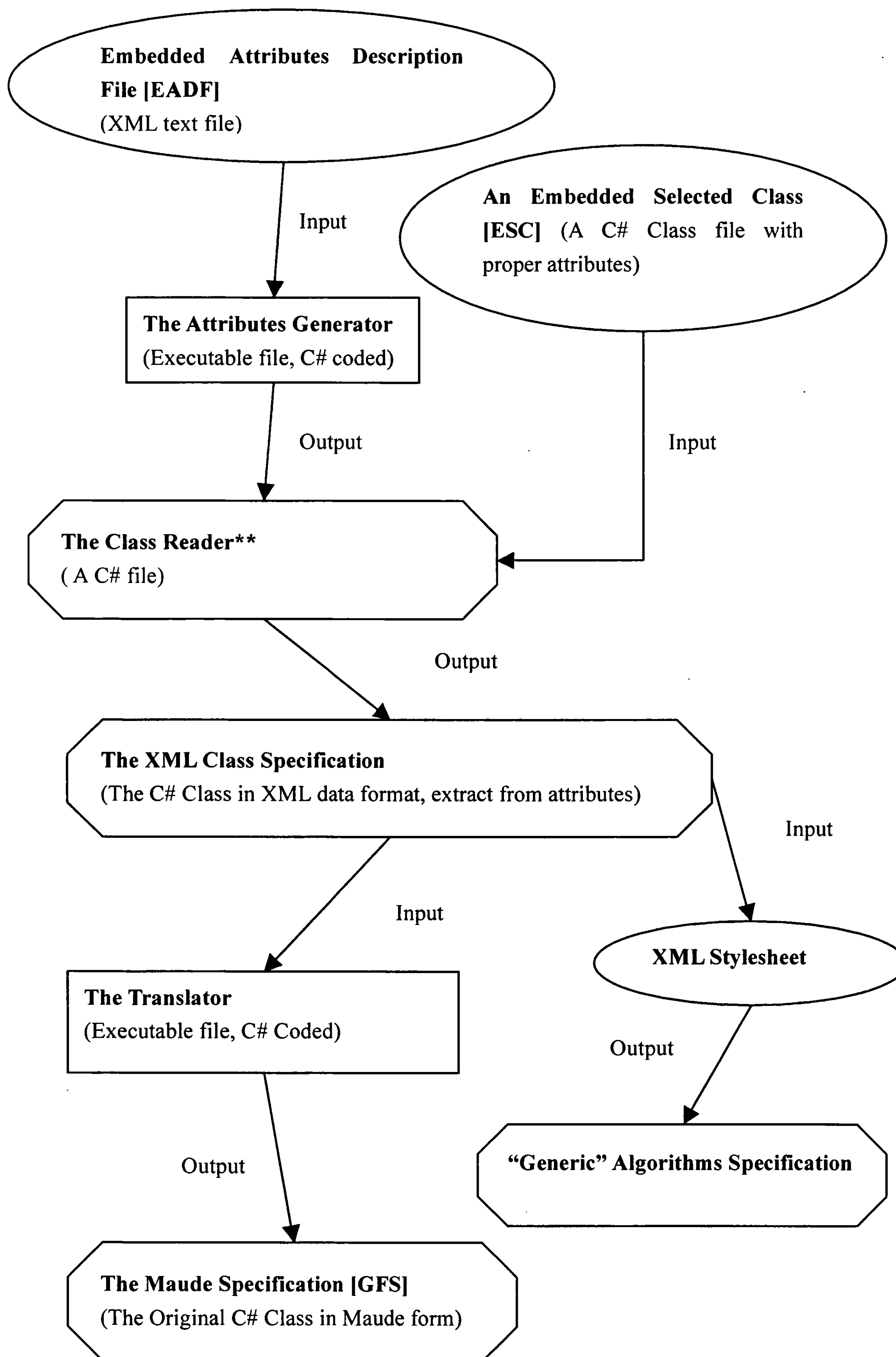


Figure3_5: The Whole Structure of the Research at the current stage

Chapter 4 Embedding Generic Formal Specifications and Documentation

In Chapter 3 we gave an overview of the aims and methodology of the work in this thesis. In this chapter we will explain our work in detail so that the functions of our generic framework can be understood. We will explain how we can define the required C# attributes in the EADF and how we can generate the Class Reader from the EADF. We will then explain the modelling process in mapping from a selected C# class into an XML class specification. Finally we will explain how we can model an XML class specification in Maude.

4.1 Transforming the EADF into C# Code

Here we will examine the process of creating the Class Reader from the EADF. The aim of this section is to show how to define attributes in a more generic way and how to translate them into a C# source program.

4.1.1 Defining the Attributes in XML

As we have already explained in Chapter 3, the aim is to create a highly configurable software framework for embedding formal specifications and documentation within program code. Attributes are the key element for our framework. We want to give the users the ability to modify the pre-defined attributes or create their own attributes, so that they can define specifications or documentation in a style of their choosing.

Definition: Embedded Attributes Description File
An Embedded Attributes Description file is an XML file that characterizes the attributes that define a particular formal specification/documentation style embedded in a C# program.

EADF files can be validated against the following XML schema:

```
<xsd:element name="Attribute" type="AttributeType"/>
```

```
<xsd:complexType name="AttributeType">
        <xsd:sequence>
                <xsd:element name="Name" type="xsd:string"/>
                <xsd:element ref="Compact" minOccurs="1"/>
                <xsd:element name="Constructor" type="xsd:anyType"/>
        </xsd:sequence>
</xsd:complexType>

<xsd:element name="Compact" type="CompactType"/>

<xsd:element name="Compact">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="Type">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                            <xsd:enumeration value="accessor" />
                            <xsd:enumeration value="parameter" />
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

The schema describes the following component parts of the EADF file:

- Name of the attributes, type is string and is unique
- Parameters of the attributes, and there are at least 1 parameter for an attribute. And there are two different types of the parameters.
- Constructor of the attributes.

The following example shows how we define the two attributes (*Hidden* and *MethodComment*) in EADF format:

```
<Attribute>
    <Name>Hidden</Name>
    <Compact Type="accessor">operation</Compact>
    <Compact Type="parameter">comment</Compact>
    <Constructer />
</Attribute>
<Attribute>
```

```
        <Name>MethodComment</Name>
        <Compact Type="parameter">mtype</Compact>
        <Compact Type="parameter">comment</Compact>
        <Constructer />
    </Attribute>
```

Example4_1: Attribute Hidden and MethodComment in XML definition

As we can see in Example4_1, each attribute must be defined by starting with the tag *<Attribute>* and closed with *</Attribute>*. And we require each attribute to have 3 types of elements which are restricted in order to generate the attribute class. The three elements are: *<Name>, <Compact>, <Constructor>*. We could use DTD or XML schemas to validate the data for these attributes' definition.

The tag *<Name>...</Name>* encloses the name of the attribute class, such as *Hidden* and *MethodComment* in the above code. Each attribute should only have one unique name, since this unique name will be the generated C# attribute class name. For example, the following is the attribute class generated from Example4_1:

```
public class Hidden : System.Attribute
{...   }
public class MethodComment : System.Attribute
{..    }
```

The tag *<Compact>...</Compact>* encloses the fields of the attribute class, which has two different types: "accessor" and "parameter". The type "accessor" means this variable is possibly present as a parameter of the constructor of the attribute class and it must be a read-only parameter. The type "parameter" means this variable is optionally a parameter of the constructor of the attribute class and its value could be set by the user. Of course, a class in C# could have multiple parameters, so we allow users to be able to define multiple elements *<Compact>* and the corresponding types must be set properly. In the attribute *Hidden* of Example4_1, we have:

```
<Compact Type="accessor">operation</Compact>
<Compact Type="parameter">comment</Compact>
```

And similarly in the attribute *MethodComment*, we have:

```
<Compact Type="parameter">mtype</Compact>
<Compact Type="parameter">comment</Compact>
```

We translate an EADF file into C# code by using the Attributes Generator. The

following is the result generated[3] from Example4_1 (Detail of the modelling also see chapter 4.1.3 and Chapter 4.2.1):

```
public class Hidden: System.Attribute
{
string operation;
string comment;

public Hidden
(string operation)
{
this.operation = operation
}

public string Aoperation
{
get
{
return operation
}
}

public string Acomment
{
get
{
return comment
}
set
{
comment = value;
}
}
}

public class MethodComment: System.Attribute
{
string mtype;
string comment;

public MethodComment
```

[3] The automatically generated C# source code is not designed for human reading since it would not contain any Comments and would not be well formatted. The purpose of the example here is only to show what the generated C# source code looks like.

```
()
{
}

public string Amtype
{
get
{
return mtype
}
set
{
mtype = value;
}
}

public string Acomment
{
get
{
return comment
}
set
{
comment = value;
}
}
}
```

Attributes *Hidden* and *MethodComment* are very simple examples. A C# class could have many methods, fields and a complicated constructor to provide various functions. The following example is the attribute *Eq* used in the algebraic specification examples in this thesis:

```
<Attribute>
    <Name>Eq</Name>
    <Compact Type="accessor">equation</Compact>
    <Compact Type="accessor">condition</Compact>
    <Compact Type="accessor">lhs</Compact>
    <Compact Type="accessor">rhs</Compact>
    <Compact Type="parameter">mtype</Compact>
    <Compact Type="parameter">comment</Compact>
    <Constructer>
        int isCondition=0;
```

```
        int ix;

        isCondition = equation.IndexOf(":");
        if (isCondition==-1)
        {
          this.condition = "NULL";
        }
        else
        {
          this.condition = equation.Substring(isCondition+1);
          equation = equation.Substring (0,isCondition);
        }
        this.equation=equation;
        ix=equation.IndexOf("==");
        this.lhs = equation.Substring(0,ix).Trim(' ');
        this.rhs = equation.Substring(ix+2).Trim(' ');
        </Constructer>
    </Attribute>
```

Example4_2: Attribute Eq in XML definition

The tag *<Constructor>...</Constructor>* simply encloses the code which is manually written by the user and will be copied as the content of the constructor of the attribute class. The reason we leave the format of constructors open is, the C# code for it is varied and hard to predict.

This is obviously not a completely satisfactory solution and we would prefer a method of defining constructors that more closely matches the way we define other components. However, we have designed the format to allow the current representation of constructors to be straightforwardly replaced (see chapter 4.1.2). The following code for the attribute class *Eq* is also generated by the Attribute Generator from Example4_2 (as we explained in Chapter 3.4, the generator automatically generate the names like "Acondition"):

```
public class Eq: System.Attribute
{
string equation;
string condition;
string lhs;
string rhs;
string mtype;
string comment;

public Eq
(string equation; string condition; string lhs; string rhs)
```

```
{

        int isCondition=0;
        int ix;

        isCondition = equation.IndexOf(":");
        if (isCondition==-1)
        {
          this.condition = "NULL";
        }
        else
        {
          this.condition = equation.Substring(isCondition+1);
          equation = equation.Substring (0,isCondition);
        }
        this.equation=equation;
        ix=equation.IndexOf("==");
        this.lhs = equation.Substring(0,ix).Trim(' ');
        this.rhs = equation.Substring(ix+2).Trim(' ');

}

public string Aequation
{
get
{
return equation
}
}

public string Acondition
{
get
{
return condition
}
}

public string Alhs
{
get
{
return lhs
}
```

```
}

public string Arhs
{
get
{
return rhs
}
}

public string Amtype
{
get
{
return mtype
}
set
{
mtype = value;
}
}

public string Acomment
{
get
{
return comment
}
set
{
comment = value;
}
}
}
```

As we already stated in Chapter 3, the member names (e.g. Aequation) are automatically generated also, and hence not chosen to be easy for humans to read. Also it should be noted the constructor *Eq* has more parameters than we intended, which is the limitation of the usage of type "accessor" (see solution in Chapter 4.1.2).

Note that <Constructor> can be an empty element, like the one in the attribute Hidden. In this case, we would automatically generate code for the constructor of the attribute class in the following format:

```
public Hidden
        (string operation)
    {
        this.operation = operation;
    }
```

One of our aims in this project is to build generic tools that would be easily read and modified by the user. The use of code embedded in the element *<Constructor>* is something that should ultimately be replaced by XML.

4.1.2 Handling Constructors: An Extended XML Format

A significant drawback of our existing XML format is that constructors are not well handled. It is not convenient to directly embed the constructor source code in the EADF file. In this section, we extend our XML format for EADF to fix this problem. The extension we propose is able to accommodate all the examples in this thesis. It is possible that other, more complex, examples would not be adequately handled. However, the XML format could be extended further if necessary.

An example of the extended format is as follows:

```
<?xml version="1.0"?>
<ClassReader>
    <ESC_Name>stackOfElt</ESC_Name>
    <Attribute type="Class">
        <Name>Hidden</Name>
        <Compact Type="construct">operation</Compact>
        <Compact Type="parameter">comment</Compact>
        <Constructor />
    </Attribute>
    <Attribute type="Method">
        <Name>Eq</Name>
        <Compact Type="construct">equation</Compact>
        <Compact Type="accessor">condition</Compact>
        <Compact Type="accessor">lhs</Compact>
        <Compact Type="accessor">rhs</Compact>
        <Compact Type="parameter">mtype</Compact>
        <Compact Type="parameter">comment</Compact>
```

```
        <Constructor>
            <Separator>==</Separator>
        </Constructor>
    </Attribute>
    <Attribute type="Method">
        <Name>Eqs</Name>
        <Compact Type="construct">equations</Compact>
        <Compact Type="accessor">condition</Compact>
        <Compact Type="array">eqArray</Compact>
        <Compact Type="parameter">mtype</Compact>
        <Compact Type="parameter">comment</Compact>
        <Constructor>
            <Separator>;</Separator>
        </Constructor>
    </Attribute>
    <Attribute type="Field">
        <Name>Vars</Name>
        <Compact Type="construct">vars</Compact>
        <Compact Type="array">varArray</Compact>
        <Constructor>
            <Separator>;</Separator>
        </Constructor>
    </Attribute>
    <Attribute type="Struct">
        <Name>Varstruct</Name>
        <Compact Type="construct">var</Compact>
        <Compact Type="construct">varType</Compact>
        <Constructor />
    </Attribute>
    <Attribute type="Struct">
        <Name>Eqstruct</Name>
        <Compact Type="construct">equation</Compact>
        <Compact Type="accessor">lhs</Compact>
        <Compact Type="accessor">rhs</Compact>
        <Constructor>
            <Separator>==</Separator>
        </Constructor>
    </Attribute>
    <ESC_Path>C:\ESC.txt</ESC_Path>
</ClassReader>
```

Example4_3: New Prospect of EADF format

Comparing Example4_3 with Example4_1 and Example4_2, we can distinguish the differences by the addition of several new tags:

- Attributes Types. These indicate the location of the attributes in the object-orient class, i.e. "Class", "Method" and "Fields". The type "Struct" tells us the attribute is an abnormal one and works as an array of storage.
- New Compact Type "construct". This means this variable must be present as a parameter of the constructor of the attribute class and it is a read-only parameter. Hence the type "accessor" is no longer considered a possible parameter for the constructor of the class and is now only read-only parameters.
- New Compact Type "array". This means the parameter is a *struct* type in C# (or say an array). It links the new attribute type "struct", which will not only store an array of the content but may also work as a parser (see below tag *<Separator>*).
- New Parsing Tag *<Separator>*. This tells the program what "expression" we are using as a separator string to parse the content (normally a string type) in the attributes. Also it removes all the C# code in the Example4_2.

The tag *<ESC_Name>* and *<ESC_Path>* tells the class name of the C# class and the location of the source code.

With this new approach of the definition of the EADF, we are definitely able to generate a fully executable Class Reader if the time permits. This new format enables us to encode all the constructors used in our examples, without resorting to directly embedding the source code.

The new EADF file (Example4_3) can be validated against the following XML schema:

```
<xsd:element name="ClassReader" type="ClassReaderType"/>

<xsd:complexType name="ClassReaderType">
        <xsd:sequence>
                <xsd:element name="ESC_Name" type="xsd:string" />
                <xsd:element ref="Attribute" minOccurs="1"/>
                <xsd:element name="ESC_Path" type="xsd:string" />
        </xsd:sequence>
</xsd:complexType>

<xsd:element name="Attribute" type="AttributeType"/>

<xsd:complexType name="AttributeType">
        <xsd:sequence>
                <xsd:element name="Name" type="xsd:string"/>
                <xsd:element ref="Compact" minOccurs="1"/>
                <xsd:element name="Constructor">
```

```
                    <xsd:complexType>
                        <xsd:element name="Separator" type="xsd:string" minOccurs="0"/>
                    </xsd:complexType>
        </xsd:sequence>
        <xsd:attribute name="type">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="Class" />
                    <xsd:enumeration value="Method" />
                    <xsd:enumeration value="Field" />
                    <xsd:enumeration value="Struct" />
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
</xsd:complexType>

<xsd:element name="Compact">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="Type">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                            <xsd:enumeration value="construct" />
                            <xsd:enumeration value="accessor" />
                            <xsd:enumeration value="array" />
                            <xsd:enumeration value="parameter" />
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

The schema describes the following component parts of the EADF file:

- *<ClassReader>* has three components: *<ESC_Name>*, *<Attribute>* and *<ESC_Path>*.
- *<Attribute>* has three components: *<Name>*, *<Compact>* and *<Constructor>*. It also has four different types: *Class, Method, Field* and *Struct*.
- *<Compact>* has four different types: *construct, array, accessor* and

parameter. *<Attribute>* can have multiple sub-elements *<Compact>*.
- *<Constructor>* can be an empty element or has one sub-element *<Separator>*.

We have noted in Chapter 4.1.1 that the left and right-hand sides of equations are simply unparsed strings, and that a better solution would be to represent equations as parse trees. The current XML format does not permit such a representation to be defined. There is no reason that our XML format cannot be extended to permit the definition of arbitrary grammars. However the need to represent expressions (in a typical object-oriented programming C-style language syntax style) is likely to be sufficiently common that it may be more appropriate to hard-code expression parsing into our program, and simply add a new XML 'expression' tag.

4.1.3 The Attributes Generator

In this section, we consider the Attributes Generator. The generator is an executable utility which reads the data from the EADF and outputs a well-constructed C# application which is known as the Class Reader.

We can divide the generator into three parts:
- The attribute-classes generator
- The ESC generator*(future development)
- The main-code generator (future development)

The Attributes Generator processes an EADF file line by line and makes decisions to generate the proper data that we need. We will not explain the source code of this generator in details since it is not a complex application.

The ESC generator is functioning as its name explains, and generates the ESC from the EADF. Here we need to note that, for this generator, we need to write the ESC in the XML format within the EDAF. It may seem to be a bit complicated compared with what we could do in Figure4_1, but writing the ESC in the XML format makes our framework become a language independent generic tool which can be adapted to a range of object-oriented languages and not only C#. We will explain this Generator in more detail in Chapter 6. However, this Generator is not essential here since the transformation process could be done in some other way. An alternative way is to replace this generator with a simple windows program with user interface as in Figure4_1. Users may choose a C# class and embed it with proper attributes. The button Modify will combine these codes with the pre-generated Class Reader source code. We believe this is a good and easy alternative, however, we still prefer to try to put the ESC within

the EADF. Then the users can define the attributes at the beginning of the EADF and embed them into the C# Class at the end of EADF. For now, we will just leave the ESC in C# format as we explained in Chapter 3.

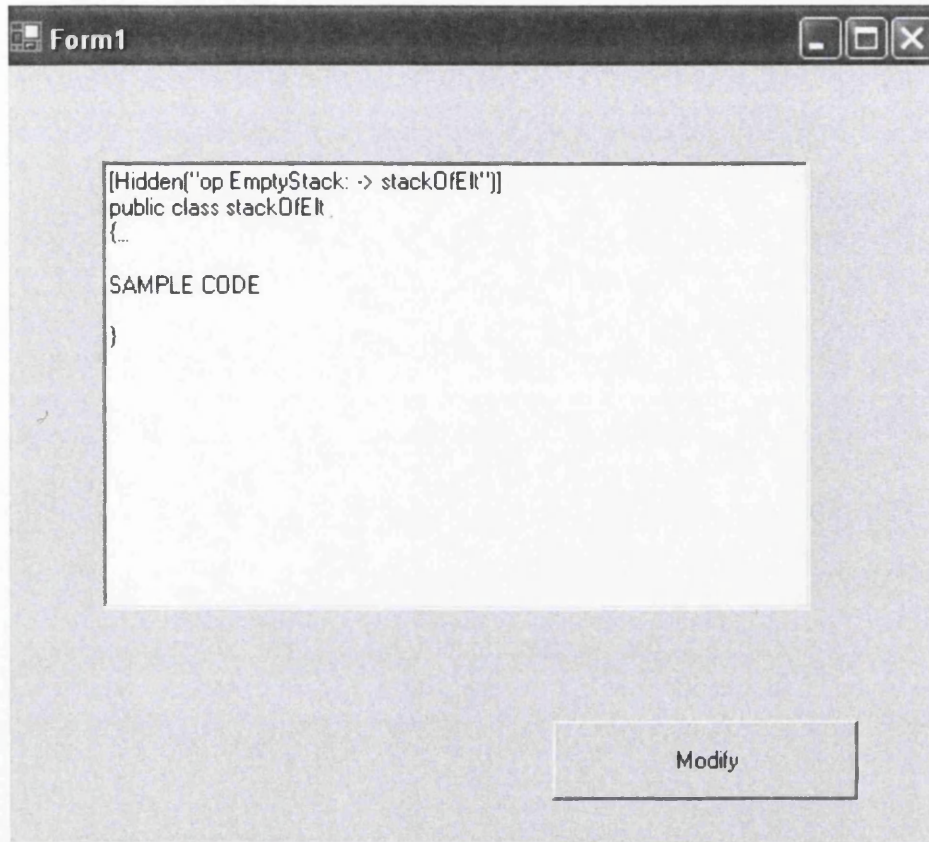


Figure4_1: Example of Writing ESC in a form application

The main-code generator is currently not implemented. First, we need to know how many attributes there are, and how many parameters each attribute has. Second, the format of the code we are going to generate is not that regular and is hard to normalize in a simple way (see more actual code of Class Reader in Chapter 4.2.1). We will leave it to a future study, however, there should be a solution and it is an important part of making the whole system more generic.

As the two parts of this generator is not properly running, we will consider the generator is an "Attributes Generator" in this dissertation instead of a "ClassReader Generator".

4.2 Modelling the ESC into XML Class Specification

In this part, we will examine the structure of the Class Reader, the process of

translating an embedded specification C# class (ESC) into the formal specification in XML data format. This XML class specification could be considered as a "universal" Algebraic formal specification notation (for the specific examples we have considered so far) which can be transformed into other specific Algebraic languages, such as Maude which we will introduce in the next chapter. Furthermore, by constructing a different set of attributes, we can develop other "universal" specification notations. For example, see Chapter 6.

4.2.1 The Structure and the Syntax of the Class Reader

The Class Reader[4] is divided into two parts. The first part contains all the necessary attributes classes, and the second part reads the embedded specification from the selected C# class by using the reflection and outputs it into an XML Class specification.

In order to give a good idea what this Class Reader looks like and how it works, we take a look at C# *stackOfElt* class which has some common methods like *pop()*, *top()* and *push()* and consider how to algebraically specify it.

As we already mentioned in the previous chapter, we need to tell the compiler which kind of elements our attributes can be used with, *AttributeUsage* does this:

```
[AttributeUsage(AttributeTargets.Class |
 AttributeTargets.Constructor |
 AttributeTargets.Field |
 AttributeTargets.Method |
 AttributeTargets.Property,
 AllowMultiple = true)]
```

Now we need to create all the attribute classes required for the C# *stackOfElt* class, which includes: *Hidden* (attribute for hidden object), *MethodComment*, *Eq* (attribute for a single equation), *Eqs* (attribute for multiple equations), *CEq* (attribute for a conditional equation), and *Vars* (attribute for variables). Let us consider the class *Hidden*:

```
//attributes for hidden object
public class Hidden : System.Attribute
{
    //private member data
```

[4] The Class Reader here is the version of manually written since the code will be easier to read and in the good programming style. Also the automatically generated version of the Class Reader is still in Alpha and could not be executed smoothly at the moment (which we have already explained in Chapter 4.1.3).

```
    private string operation;
    private string comment;

    //attribute constructor for positional parameters
    public Hidden
        (string operation)
    {
        this.operation = operation;
    }

    //accessor, read-only properties for the positional parameters
    public string Operation
    {
        get
        {
            return operation;
        }
    }

    //Named parameters are implemented as properties
    public string Comment
    {
        get
        {
            return comment;
        }
        set
        {
            comment = value;
        }
    }
}
```

To define each attribute class, we need to extend *System.Attribute*, and each attribute class would have its own private parameters (or fields) and constructor. The constructor should specify which parameters must be in the constructor. The parameters must be written in the order declared in the constructor. We call these parameters the *positional* parameters, in the above *Hidden* class, *Operation* is the positional parameter. The other parameter *Comment* is implemented as properties, and is *a named parameter*. Named parameters are not necessarily in the constructor and they are normally working as commentary.

Since all the parameters are private members of the attribute class, in order to

access them, we need to add *accessor* methods. Function *get()* will return the proper value of the parameters we require for access and function *set()* will give the proper value to the parameters we require to access. They are both pre-defined in C#. Positional parameters are normally read-only so that they may only have *get()* accessor methods. Named parameters normally have both *get()* and *set()* accessor methods.

The *Hidden* attribute class is one of the simplest in our algebraic example. Others, like class *Eq* and *CEq*, are much more complicated if we consider their constructors and methods in details, but they all have a similar structure.

```
//attributes for equations
    public class Eqs : System.Attribute
    {
        //private member data
        private string equations;
        private string condition;   //condition part of eq.
        private string comment;
        private eqStruct[] eqArray;
        private string mType; //method type

        //attribute constructor for positional parameters
        public Eqs
            (string equations)
        {
            int isCondition=0;

            isCondition = equations.IndexOf(":");
            if (isCondition==-1)
            {
                this.condition = "NULL";
            }
            else
            {
                this.condition = equations.Substring(isCondition+1);
                equations = equations.Substring (0,isCondition);
            }
            this.equations=equations;

            int i=0;
            foreach (string eq in equations.Split(';'))
            {
                i++;
            }
```

```
        eqStruct[] eqArray;
        eqArray = new eqStruct[i];
        i=0;
        foreach (string eq in equations.Split(';'))
        {
            eqArray[i] = new eqStruct(eq);
            i++;
        }
        this.eqArray = eqArray;

    }

    //accessor, read-only properties for the positional parameters
    public string Equations
    {
        get
        {
            return equations;
        }
    }

    //accessor
    public string Condition
    {
        get
        {
            return condition;
        }
    }

    public eqStruct[] EqArray
    {
        get
        {
            return eqArray;
        }
    }

    //Named parameters are implemented as properties
    public string Comment
    {
        get
        {
```

```
                return comment;
            }
            set
            {
                comment = value;
            }
        }

        //Named parameters are implemented as properties
        public string MType
        {
            get
            {
                return mType;
            }
            set
            {
                mType = value;
            }
        }

    }
```

Now we consider the main code. The main code is principally about how to create a new XML file by extracting the embedded equational specification which is partly defined by the actual C# code that is being documented/specified and partly by the embedded attributes.

We have already introduced Reflection in C#. Here we would like to use it to read the metadata in the *stackOfElt* class. We initialized the object *inf* of the type *MemberInfo* on the *stackOfElf* type, which would return us the necessary information about the members of the class, such as methods and fields.

```
//get the member information and use it to retrieve the custom attributes
System.Reflection.MemberInfo inf =typeof(stackOfElt);
```

Then we need to call *GetCustomAttributes* on the object *inf*, which would return us an array of objects of a specified attribute. For example, for the attribute *Hidden*, we have:

```
object[] attHidden;
attHidden = inf.GetCustomAttributes(typeof(Hidden),false);
```

Array *attHidden* contains all the objects we require, and now we need to iterate

over the array to retrieve the relevant properties. Also, we need to output it in the proper XML format:

```
//print out the hidden object on the xml file
foreach (Object attribute in attHidden)
{
    Hidden hd = (Hidden) attribute;
    //output hidden: <Hidden>
    writer.WriteStartElement("","Hidden","");

    //output operation: <Operation>... </Operation>
    writer.WriteStartElement("","Operation","");
    writer.WriteString(hd.Operation);
    writer.WriteEndElement();

    //output comment: <Comment>... </Comment>
    writer.WriteStartElement("","Comment","");
    writer.WriteString(hd.Comment);
    writer.WriteEndElement();

    //end hidden tag: </Hidden>
    writer.WriteEndElement();
}
```

The code about attribute *Hidden* is simple but also very specific as it is the unique attribute of the class. But how about the attributes embedded inside the class *stackOfElt*? Any class may have more than one field and may also have more than one method. Then we must search them one by one through the whole class.

```
//field object
foreach(FieldInfo fInfo in typeof(stackOfElt).GetFields())
{
    object[] attVars;
    attVars = fInfo.GetCustomAttributes(typeof(Vars), false);

    foreach (Object attribute in attVars)
    {
        Vars va = (Vars) attribute;
        //output vars
        foreach (varStruct e in va.VarArray)
        {
            writer.WriteStartElement("","Field","");
            writer.WriteStartAttribute("","Type","");
```

```
                writer.WriteString(e.VarType);
                writer.WriteEndAttribute();
                writer.WriteString(e.Var);
                writer.WriteEndElement();
            }
        }
    }
```

The code for finding attribute *Vars* inside the class *stackOfElt* is a little different from attribute *Hidden*. We need to retrieve the information from the object *finfo* of the type *FieldInfo*, which provides access to the metadata of the fields of the class *stackofElt*. And here we have to call *GetFields()* on the *typeof* operator in order to get the specified metadata we actually need. As we know, all the parameters in the class have specified types, for example: String *myName*, String is the type of *myName*. Therefore we have a struct *varStruct* pre-defined in attribute class *Vars*, which can store both the value and the type of the parameters of the class.

Similarly to the attribute *Vars*, we have the following code to access the metadata we need from the attributes of methods.

```
//methods
foreach(MethodInfo mInfo in typeof(stackOfElt).GetMethods())
{... }
```

Although all the attributes of the methods like *MethodComment, Eq* and *CEq*, contain totally different metadata, the structure and the syntax of the codes to access this information is almost the same. Let us have a look at the attribute *MethodComment*:

```
object[] attMethodComment;
attMethodCommen t=
mInfo.GetCustomAttributes(typeof(MethodComment),false);

//iterate through the attributes, retrieving the properties
//hidden object
foreach (Object attribute in attMethodComment)
{
    MethodComment MC = (MethodComment) attribute;
    //output hidden
    writer.WriteStartElement("","Method","");
    writer.WriteStartAttribute("","Type","");
    writer.WriteString(MC.MType);
```

```
        writer.WriteEndAttribute();

        //output method name
        writer.WriteStartElement("","Name","");
        writer.WriteString(mInfo.Name);
        writer.WriteEndElement();

        //output parameter type of this method
        writer.WriteStartElement("","ParameterType","");
        foreach(ParameterInfo PInfo in mInfo.GetParameters())
        {
            writer.WriteString(PInfo.ParameterType.ToString());
        }
        writer.WriteEndElement();

        //output the type of the method
        writer.WriteStartElement("","MethodReturnType","");
        writer.WriteString(mInfo.ReturnType.ToString());
        writer.WriteEndElement();

        //output comment
        writer.WriteStartElement("","Comment","");
        writer.WriteString(MC.Comment);
        writer.WriteEndElement();

        //end hidden tag
        writer.WriteEndElement();
    }
```

In C#, all metadata about methods – names, parameters, return type etc. -is accessible using Reflection. In order to access the parameters of the methods, we need to create a new object *PInfo* of the type *ParameterInfo*. As we can see in the above code, we already have the object *mInfo* to grant access to the metadata of the current method. Then we can call *GetParameters()* on this *MethodInfo* object to get back the type of the parameter:

```
foreach(ParameterInfo PInfo in mInfo.GetParameters())
{
        writer.WriteString(PInfo.ParameterType.ToString());
}
```

We have already looked through the whole program of the Class Reader. On the way of developing this software utility, we wanted to automatically generate the whole code of this part, but due to the complexity of the main code, we stopped

at the midway and leave it to the future research. Since we have already known the code of translating the ESC into a XML class specification, we will have a close look at the both side.

4.2.2 Modelling the ESC into XML Class Specification

In this project, our program should able to pick any C# Class embedded with attributes and transform it into XML class specification. Since we have already used class *stackOfElt* on the above examples, we shall continue to use it here to make things easy to understand. Consider the class *stackOfElt* first (which we have introduced in chapter 1 and previous sections):

```
//an example code of a int stack
    [Hidden("op EmptyStack : -> stackOfElt")]
    public class stackOfElt
    {
        [Vars("E:Int;S:stackOfElt")]
        public int Elt;
        public Stack myStack;

        //constructor--create an empty stack
        public stackOfElt()
        {
            ...
        }

        //constructor--create an empty stack with a element
        public stackOfElt (int Elt)
        {
            ...
        }

        [Eq("((S .push(E)) .s) .top()== E .",MType="Query",Comment="S is myStack, E is
Element.")]
        public int top()
        {
            ...
        }
```

```
        [Eq("((S .push(E)) .s) .pop()== S .",MType="Command",Comment="S is myStack, E is Element.")]
        public void pop()
        {
            ...
        }

        [Eq("((S .push(E)) .s) .pop()== S .",MType="Command",Comment="S is myStack, E is Element.")]
        public void push(int Elt)
        {
            ...
            }
        }

        [Eqs("((S .push(E)) .s) .topAndpop() .q == E .;((S .push(E)) .s) .topAndpop() .s == S .",
              MType="Query and Command",Comment="S is myStack, E is Element.")]
        public int topAndpop ()
        {
            ...
        }
    }
```

Attributes make the process of embedding and extracting information about a C# class straightforward. The attributes we have in this C# class are: *Hidden, Vars, MethodComment* and *Eqs*. They are written by users (not necessarily the same users who will embed and use them in C# classes) and all the data in the attributes is currently of the type string and must be arranged in the proper order:

Hidden [operation, comment]
Vars [fields or parameters]
MethodComment [MType, comment]
Eqs [equations, MType, comment]

The "operation" in the attribute Hidden is in the format of an equation in Maude. (Note that a more sophisticated approach would structure the expressions on either side of each equation using XML tags, rather than simply representing them as text strings. However, since this would require parsing the equations, we leave this as a future enhancement. Note that this is means that the current implementation does not verify the equation syntax.) The "comment" is not required in the attributes but it enables an explanation to be given of what this method is for, which makes the attributes easier to understand. The "parameters" and "equations" are strings here, but they need to be in the

proper format shown in the example code since they contain important information we need to do the transformation. A common classification concept for methods in Object Oriented Programming is to distinguish between *Commands* – which are methods that modify objects and Queries – which are methods that return information about objects without modifying them. Some methods both modify objects and return information and hence fall into both categories. Information about the classification of a method in this style could be useful when defining its specification. For example, in the case of algebraic specification, the defining equations for Queries and Commands are simpler than for methods which are both. This is because methods that are both Queries and Commands must return data pairs – representing the new value of the object and the value of the query. We use *MType* to represent a methods' type. Normally if the method does not have a return value (with a return type of Void), then we say *MType* is "Command". Otherwise we can say its *MType* is "Query", or "Query and Command" depending on whether the method simply returns information or additionally modifies an object's state. As we already stated in the earlier stage, these attributes could be customised in the way the users liked, and as far as possible we avoid forcing specific decisions on users. We also try to avoid restrictions that would prevent C# being replaced by another object oriented language (see more about Methods and MType in Chapter 4.3.5).

Now we consider the XML notation that we have developed to model the C# class. The XML file contains all the necessary information we need later on to create an executable specification in Maude, and it is automatically generated in the very strict format by the Class Reader. Here is a simple example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="classSpec.xsl"?>
<Class>
        <Name>stackOfElt</Name>
        <Extend>stack</Extend>
        <Hidden>
                <Operation>op EmptyStack : -> stackOfElt</Operation>
                <Comment />
        </Hidden>
        <Field Type="Int">E</Field>
        <Field Type="stackOfElt">S</Field>
        <Method Type="Query">
                <Name>top</Name>
                <ParameterType />
                <MethodReturnType>System.Int32</MethodReturnType>
                <Equation>
                        <LHS>((S .push(E)) .s) .top()</LHS>
```

```xml
                <RHS> E .</RHS>
            </Equation>
            <Condition>NULL</Condition>
            <Comment>S is myStack, E is Element.</Comment>
        </Method>
        <Method Type="Command">
            <Name>pop</Name>
            <ParameterType />
            <MethodReturnType>System.Void</MethodReturnType>
            <Equation>
                <LHS>((S .push(E)) .s) .pop()</LHS>
                <RHS> S .</RHS>
            </Equation>
            <Condition>NULL</Condition>
            <Comment>S is myStack, E is Element.</Comment>
        </Method>
        <Method Type="Command">
            <Name>push</Name>
            <ParameterType>System.Int32</ParameterType>
            <MethodReturnType>System.Void</MethodReturnType>
            <Equation>
                <LHS>((S .push(E)) .s) .pop()</LHS>
                <RHS> S .</RHS>
            </Equation>
            <Condition>NULL</Condition>
            <Comment>S is myStack, E is Element.</Comment>
        </Method>
        <Method Type="Query and Command">
            <Name>topAndpop</Name>
            <ParameterType />
            <MethodReturnType>System.Int32</MethodReturnType>
            <Equation>
                <LHS>((S .push(E)) .s) .topAndpop() .q </LHS>
                <RHS> E .</RHS>
            </Equation>
            <Equation>
                <LHS>((S .push(E)) .s) .topAndpop() .s </LHS>
                <RHS> S .</RHS>
            </Equation>
            <Condition>NULL</Condition>
            <Comment>S is myStack, E is Element.</Comment>
        </Method>
</Class>
```

The whole class is enclosed by tag *<Class>* and *</Class>*。Similarly, the methods are enclosed by the tag *<Method>* and *</Method>*, and the equations are enclosed by the tag *<Equation>* and *</Equation>*. There is a "*Type*" in each tag *<Method>* to determine whether this method is "*Query*" or "*Command*" or both. And each equation has been divided into "*LHS*" (left hand side) and "*RHS*" (right hand side), in order to make the transformation easier from XML class specification to Maude specification. Tag *<Condition>...</Condition>* identifies if it is a conditional equation. Because this XML class specification is generated automatically, we can see some C# type declarations like "*System.Void*" and "*System.Int32*" in this XML file. All the tags and data in this XML class specification are clear and easy to understand, but it is still not that convenient for us to check if this file contains all the information we required from the original C# class. We can use an XML schema to define the structure of this XML specification file:

Definition

The XML Class Specification for C# class *stackOfElt* is syntactically defined by the following XML Schema:

```
<xsd:element name="Method" type="MethodType"/>

<xsd:complexType name="MethodType">
        <xsd:sequence>
                <xsd:element name="Name" type="xsd:string"/>
                <xsd:element ref="ParameterType" minOccurs="0" maxOccurs="10"/>
                <xsd:element name="MethodReturnType type" type="xsd:string"/>
                <xsd:element name="Comment" type="xsd:anyType"/>
                <xsd:element ref="Equation" minOccurs="0" maxOccurs="10"/>
                <xsd:element ref="Condition" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="Type">
                <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="Query" />
                                <xsd:enumeration value="Command" />
                                <xsd:enumeration value="Query and Command" />
                        </xsd:restriction>
                </xsd:simpleType>
        </xsd:attribute>
</xsd:complexType>

<xsd:element name="ParameterType" type="xsd:string"/>

<xsd:element name="Equation" type="EquationType"/>
```

```
<xsd:complexType name="EquationType">
    <xsd:sequence>
        <xsd:element name="LHS" type="xsd:string"/>
        <xsd:element name="RHS" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="Condition" type="xsd:string"/>
```

Here we need to note that the schema would be different for a different XML class specification (the names of the attributes are defined by the users, and different attributes will generate different XML tags). At the moment it is not clear how to generically define the output as a schema: in fact, no such `meta schema' definition method exists at all at the moment. Anecdotal accounts of attempts to define "generic meta schemas" so far seem to result in highly abstract definitions of limited usefulness. An alternative (and possible future extension) would be for the system to automatically output the schemas in conjunction with class specifications.

XML has offered us a mechanism to make the XML file transform into a simple class specification. The utility is Extensible Stylesheet Langugage (XSL). After we add the XSL stylesheet in our XML file: *<?xml-stylesheet type="text/xsl" href="classSpec.xsl"?>*, we have the following:

```
Class stackOfElt :: stack Specification {
Int: E
stackOfElt: S
Hidden {
op EmptyStack : -> stackOfElt
//
}
Method top : -> System.Int32 [Query]{
eq ((S .push(E)) .s) .top() = E .
IF ( NULL )
//S is myStack, E is Element.
}
Method pop : -> System.Void [Command]{
eq ((S .push(E)) .s) .pop() = S .
IF ( NULL )
//S is myStack, E is Element.
}
Method push : System.Int32 -> System.Void [Command]{
eq ((S .push(E)) .s) .pop() = S .
```

```
IF ( NULL )
//S is myStack, E is Element.
}
Method topAndpop : -> System.Int32 [Query and Command]{
eq ((S .push(E)) .s) .topAndpop() .q = E .
eq ((S .push(E)) .s) .topAndpop() .s = S .
IF ( NULL )
//S is myStack, E is Element.
}
}
```

The stylesheet code is shown below:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
<xsl:apply-templates select="Class"/>
</xsl:template>

    <xsl:template match="Class">
    <P>
    <B>Class </B>
        <xsl:value-of select="Name"/>
        Specification {
        </P>
        <xsl:for-each select="Field">
        <P>
        <xsl:value-of select="./@Type"/>:
        <xsl:apply-templates/>
        </P>
        </xsl:for-each>
        <xsl:for-each select ="Hidden">
        <P>
        <B>Hidden </B>{
        </P>
        <P>
        <xsl:apply-templates/>
        </P>
        <P>
        }
        </P>
        </xsl:for-each>
```

```
        <xsl:for-each select ="Method">
        <P>
        <B>Method   </B>
         <U><xsl:value-of select="Name"/> : <xsl:value-of select="ParameterType"/> ->
<xsl:value-of select="MethodReturnType"/> </U>
          [<xsl:value-of select="./@Type"/>]{
        </P>
        <xsl:apply-templates/>
        <P>
        }
        </P>
        </xsl:for-each>
    <P>
    }
    </P>
    </xsl:template>


    <xsl:template match="Name"/>
    <xsl:template match="ParameterType"/>
    <xsl:template match="MethodReturnType"/>


    <xsl:template match="*/Equation">
    <P>
    <I>eq </I><xsl:value-of select="LHS"/> = <xsl:value-of select="RHS"/> .
    </P>
    </xsl:template>

    <xsl:template match="*/Operation">
    <P>
    <xsl:value-of select="."/>
    </P>
    </xsl:template>

    <xsl:template match="*/Condition">
    <P>
    <I>IF </I>( <xsl:value-of select="."/> )
    </P>
    </xsl:template>

    <xsl:template match="*/Comment">
    <B style="COLOR: green">//<xsl:value-of select="."/></B>
    </xsl:template>
```

</xsl:stylesheet>

This makes the output more readable, and more in the form we would typically expect for an Algebraic Specification. This XSL stylesheet is simply searching the data we want and displaying it in a clean and tidy way. For example, for all the data enclosed in the tag *<Equation>...</Equation>*, we only want to display the equation in the normalized form, like "*eq LHS = RHS*", and the XSL stylesheet actually gives us the result we wanted.

4.3 From XML Class Specification to Maude Specification

In this section we will examine the next stage in our modelling process where we convert an XML class specification into another formal specification language – in a specific algebraic language instead of the generic "algebraic style" representation of the previous section. We have chosen Maude as the targeted language to illustrate the modelling process of our generic toolset. There are a number of languages available, but Maude is well-known and understood at Swansea. The purpose of creating a Maude specification is to provide a specification that we can execute for testing and analysis purposes. This process involves the creation of many special operations and equations which are needed to allow us to execute the specification.

4.3.1 The Translator

We created a utility to transform the XML class specification into the Maude specification automatically, which is called the translator. The translator is a C# coded program that is able to read from the XML file, and then generate operations and equations in Maude syntax. We would like to state that at the time we wrote the program and thesis, we did not think it was possible to do the transformation in XSL.However, having gained much more experience in using XML it clearly is.

It is not necessary to examine the Translator code in detail, but the process of modelling the XML class specification into Maude specification is interesting and we will discuss it in detail. In the following sections, we consider the representation of the various components of a class, with specific reference to our algebraic specification example in Maude, we will discuss the modelling process step by step and the mapping between the relevant C# entities and Maude entities.

4.3.2 Class and its Name

The identifier of a class is the *classname*. A classname is a unique name which can not be shared with any other class. In C# language, a typical example of a classname is (Actually, it would be *stackOfElt* – since C# does not use the same naming conventions as Java):

```
public class stackOfElt
{...}
```

And we would have the following format in the XML class specification:

```
<Class>
    <Name>stackOfElt</Name>
    ...
</Class>
```

Strictly there is no "classname" in Maude, the nearest equivalent s the module name. We need to create a sort to represent objects of the class. Also, we need to represent not only classes, but object instances of classes. To do this, we also need to create a new sort, with the same name as the class. Therefore we should generate a new Maude file *stackOfElt.maude* from above example:

```
fmod stackOfElt is
sort stackOfElt .
...
endfm
```

In this stage, we have the mapping:

Class → Maude Functional Module+ A New Sort

4.3.3 Fields

Fields are variables which belong to class instances. In the class *stackOfElt* there are three fields:

```
[Vars("E:int; S:Stack")]
public int Elt;
public Stack myStack;
public int Elt2
```

C# fields never map to variables in Maude. We normally need to map fields into

operations. Therefore we have the mapping:

Fields → Operations

In order to model fields in Maude we define them as two operations. We need to define equations which allow us to *get* and *set* the field value of a given object. To a user knowledge of the implementation of this is not required and will appear to them as though they are assigning and accessing fields are they normally would in C#. First we need to define the get and set operations for a field:

```
op _.field : class -> field_type .
op _.filed := _ : class field_type -> class .
```

For example, if we have field "*int e*" in the class *A*, we will have the following lines:

```
op _.e : A -> int .
op _.e:=_ : A int -> A .
```

Also these operators can be used to define properties in a similar way. For example, the "standard" equation that would be generated for the field *e* above is

```
eq (A .e := x).e == x.
```

That is, if we store the value of *x* in *e* and then look up the contents of *e*, we get *x*. In fact this equation can be very simply generated automatically (though our current implementation does not yet do so). In the case of "simple" properties, where a value is stored and accessed without any tests of its value, the process (and equation) is identical - and can also be automatically generated. However, some properties are not so simple. For example, the following property stores zero if an attempt is made to store a negative number.

```
public int e
    {
        get
        {
            return e;
        }
        set
        {
            if (x <=0)
              e = 0;
```

```
            else
                e = value;
        }
    }
```

This case is similar, but we can no longer use the single, simple equation above. Instead we need the following pair of conditional equations:

ceq (A .e := x).e == x if x > 0 .
ceq (A .e := x).e == 0 if x <= 0 .

4.3.4 Constructors

Constructors are used to create class instances. In C#, the name of the constructor is always the same as the *classname*, as is the case with Java. A class can have more than one constructor provided their signatures are different. A signature is defined by its name and its parameter list: and since the name of a constructor must be the same as its class, the parameters must be different. Two constructors differ in their signatures if they have different numbers or types of parameters. Therefore if a class has more than one constructor, then the number, order and types of input parameters entered will determine which constructor is called when a class instance is created. This is called operator overloading. Methods also have this similar attribute and we will discuss it in 4.3.5. The following are examples of constructors:

```
public class stackOfElt
    {
        ...

        //constructor--create an empty stack
        public stackOfElt()
        {... }

        //constructor--create an empty stack with a element
        public stackOfElt (int Elt)
        {... }

        ...
    }
```

Constructors are a special case here, since we can see in the previous chapters,

there are no constructors in the XML class specification. Because constructors only return the class type as there is no query part to constructors, they do not necessarily appear on the final generated Maude code[5]. In the case of a default constructor like, say, new *StackOfElt()* which returns *EmptyStackofElt*, then there *could* be no constructor in the Maude code – it could just be reduced to *EmptyStackofElt*. But it doesn't *have* to be. However, we can still demonstrate the constructors in the Maude code, for example:

op emptyStack :-> StackofElt . //A constant representing an empty stack

op new StackOfElt() : -> StackofElt .
Eq new StackOfElt() = emptyStack . //for a simple one that just makes a new stack

op new StackOfElt(_) : Elt -> StackofElt .
Var e: Elt .
Eq new StackofElt(e) = emptyStack.push(e). //for one that adds an initial element)

All constructors will return a new instance of the class initialised in some manner, defined by the constructor's equations and parameters. In the case of the default constructor with no parameters, this is generally some constant operator which symbolises the empty class or default class. This empty class is usually specified by the user in the Hidden attribute. Typically, we could still have the following mapping:

Constructor → Operation

4.3.5 Methods

In C#, methods take on the role of procedures and functions in procedural programming languages (like Pascal). In this chapter, we want to do the mapping:

C# methods → operations

A method consists of a name, a list of parameters and its return type. If a method returns no value, then its return type is void, for example:

[5] The constructors in C# and in Maude are different. The constructor in C# creates a new object, but the constructor in Maude "creates" a new sort element. For example, in Chapter 3.3, we have an example of Natural number, the equations could be: op 0 : -> Nat .[ctor] and op s : Nat -> Na t.[ctor], but 0 and s are not [ctor]. Actually the ctor – constructor in Maude is a mechanism for enumerating all the elements of a sort, which is related to but not exactly the same as a constructor in C#. In C# constructors also strictly speaking enumerate the elements of a class. However, we do not generally think of them in those terms.

```
public int top()
{... }

public void pop()
{... }
```

Methods may either solely return information about a class instance (we denote such methods *Query* as we already stated before (also see Chapter 4.2.2)), solely modify a class instance (*Command*) or do both (*Query and Command*).

```
<Method Type="Query">
        <Name>top</Name>
        <ParameterType />
        <MethodReturnType>System.Int32</MethodReturnType>
        <Comment>Return the top element of the stack</Comment>
</Method>
<Method Type="Command">
        <Name>pop</Name>
        <ParameterType />
        <MethodReturnType>System.Void</MethodReturnType>
        <Comment>Rmove the 1st element of the stack</Comment>
</Method>
```

These equations in the attributes and XML class specification, represents the purpose and the functions of the current methods. The corresponding equations for *topAndpop* are as follows:

```
<Method Type="Query and Command">
        <Name>topAndpop</Name>
        <ParameterType />
        <MethodReturnType>System.Int32</MethodReturnType>
        <Equation>
                <LHS>S.push(E).topAndPop().q</LHS>
                <RHS>E</RHS>
        </Equation>
        <Equation>
                <LHS>S.push(E).topAndPop().s</LHS>
                <RHS>S</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>S is myStack, E is Element.</Comment>
</Method>
```

Now we will show how to convert the methods from an XML class specification

into a Maude specification. As discussed before (see Chapter 4.2.2), methods can either change the state of an object, or return a value, or both. The obvious was to model this is to permit such operations to return both the result value (representing the Query) and a new instance of the object (representing the Command). But an operation in Maude can not directly model functions that returns two types. In order to do so we need to create a tuple sort of the two types that can be returned. And when it comes to the equations to define each return type, we have create separate equations defining a method's return type (query) and equations defining how a method changes the state of a class instance (command) by means of functions (*projection operators)* that extract each part of the tuple. We use q to represent the projection operator for queries and use s to represent the projection operator for commands (or states). Let us consider the method *top* in the Maude code:

op top() : -> Int .

The above Maude code is correct, but we want to change the syntax of each method to more closely reflect the object-oriented member access notation. So we have the new operator for *top*:

op_. top() : stackOfElt -> Int .

The input is a class instance variable of the class itself (i.e. an object). This allows us to use the object-oriented member access notation. We can also consider the example *pop* again:

op_. pop() : stackOfElt -> stackOfElt .

The above examples *top* and *pop* are easily to be converted since method *top* is simply a query and method *pop* is simply a command. However, there are some methods like *topAndpop* we have shown above which are more complicated. We have to change the return sort to be a tuple which contains both the state change return type and the query return type. The definition of an appropriate tuple sort only needs to be done once for each different return type. In order to model this, we will need to add a new sort which will be the appropriate tuple sort type:

sort AB
A is the classname, and B is the MethodReturnType

We use the convention that the name of the tuple sort will be the concatenation of the two individual sort names. It is also of course possible that a sort of that name will already exist (or, strictly, that a type of that name will exist in the C# code). In this thesis, we neglect this possibility – though a robust system must

be able to cope with it.

Now we apply it to the above examples and we will have the following generated code:

sort stackOfEltInt .
op (_,_) : stackOfElt Int -> stackOfEltInt .
op _.q : stackOfEltInt -> Int .[6]
op _.s : stackOfEltInt -> stackOfElt .

op _.top() : stackOfElt -> stackOfEltInt .

sort stackOfEltVoid .
sort Void .
op (_,_) : stackOfElt Void -> stackOfEltVoid .
op _.q : stackOfEltVoid -> Void .
op _.s : stackOfEltVoid -> stackOfElt .

op _.pop() : stackOfElt -> stackOfEltVoid .

As for method *topAndpop*, we already have sort stackOfEltInt created:

op _.topAndpop() : stackOfElt -> stackOfEltInt .

Notice also the two projection operators:

op _.q : stackOfEltInt -> Int .
op _.s : stackOfEltInt -> stackOfElt .

Although at the moment these operators are not defined.

We have handled methods from XML class specification into Maude code, and now we need to consider equations.

4.3.6 Equations

We have already seen the equations when we were discussing methods. Each method will be defined in terms of equations and we will copy them directly from XML class specification into Maude code. Since we have already handled equations when we converted a general C# class into an XML class specification, it would be an easy job here for us to copy them into our Maude code. We have

[6] The "_.q" and "_.s" are projection operators, for example, we could have (a,b,c) → b.

the following form for equations:

eq LHS (left hand side) = RHS (right hand side) .

We could have conditional equations too:

eq LHS (left hand side) = RHS (right hand side) if Condition .

The equations *LHS* and *RHS* and the condition are all strings because we assumed they were already in the proper format in the attributes. Note that at the moment we do not do any error checking on the format of the *RHS, LHS* and *condition* (in the case of conditional equations). This is obviously not satisfactory in the long term and ultimately the expression structure of the components of equations should be properly parsed and represented. We omit it here because it would be time-consuming to implement, but is well-understood.

A method potentially returns a pair consisting of a return value and a new state for a class instance: something which would be reflected in the equations. For example, the following are the equations of method *topAndpop* in XML class specification:

```
<Equation>
        <LHS>S.push(E).topAndPop().q</LHS>
        <RHS>E</RHS>
</Equation>
<Equation>
        <LHS>S.push(E).topAndPop().s</LHS>
        <RHS>S</RHS>
</Equation>
<Condition>NULL</Condition>
<Comment>S is myStack, E is Element.</Comment>
```

Then we would have the following equations in our Maude code:

eq ((S .push(E)).s).topAndpop().q = E .
eq ((S .push(E)).s).topAndpop().s = S .

The variables used in the equations – *S and E* - will have already been declared them in the field section.

4.3.7 Sorts, Subsorts and Hidden operators

Before we consider inheritance, we shall have a brief look at the sorts and the hidden operators. The hidden operators are used to define, for example, constants which are used in the equation definitions. For instance, in order to equationally define a stack in our example class *stackOfElt*, it is helpful to have a constant that represents an empty stack. This would be done by defining an operation:

op EmptyStack : -> stackOfElt .

The *EmptyStack* operator is not an aspect of the original class *stackOfElt* that we are modelling but is considered to be useful for creating equational definitions of the class. Hence we need some mechanism to introduce it, and we have chosen to group all such operations together, and collectively call them *hidden* operators. The *Hidden* operators are usually trivial in practice and are copied unchanged when we transform the XML class specification into Maude code.

Sorts[7] require a bit more work. In order to deal with inheritance, we have to define a subsort relating the inherited class type and the inheriting class type. Therefore the inheriting class sort is a subsort of the inherited class sort, we have the following form:

subsort InheritingClass < InheritedClass

Assume we have a C# class *stack*, and our example class *stackOfElt* is inheriting from *stack*, we would have the following line in our Maude code:

subsort stackOfElt < stack .

4.3.8 Inheritance

In this section we will consider how we model inheritance and especially how we model inherited methods in Maude. In C#, we use the following syntax to define inheritance:

Public class stackOfElt : stack
{...}

[7] Note that for the moment, we do not consider simple types but only reference types. In general, the sort type can be either a primitive data type such as an integer or a real number or it can be a reference type such as an array or even a class instance of a certain class type.

We know from above example, *stackOfElt* is the inheriting class and *stack* is the inherited class or we can say *stack* is the base class of *stackOfElt*. And we would have the following code in the XML class specification:

```
<Class>
        <Name>stackOfElt</Name>
        <Extend>stack</Extend>
...
</Class>
```

The tag *<Extend>...</Extend>* tells us the inheritance of the current class. When we transform the XML Class Specification into Maude code, we will have to check the Maude code for *stack* and decide whether there is any method overriding.

```
fmod stack is
protecting baseclass .
sort stack .

op _.top() : stack -> stackInt .

sort stackInt .
op (_,_) : stack Int -> stackInt .
op _.q : stackInt -> Int .
op _.s : stackInt -> stack .

var A : stack .

eq A .top().q = 0 .
endfm
```

As we can see, class *stack* has a method *top* and one related equation. When we are transforming the class *stackOfElt* from XML class specification into a Maude specification, we would have:

```
fmod stackOfElt is
protecting stack .
sort stackOfElt .
subsort stackOfElt < stack .
op EmptyStack : -> stackOfElt .

var E : Int .
var S : stackOfElt .
```

```
op _.top() : stackOfElt -> stackOfEltInt .

subsort stackOfEltInt < stackInt .

op _.base.top() : stackOfElt -> stackOfEltInt .
var A : stackOfElt .
eq A .base.top().q = 0 .

sort stackOfEltInt .
op (_,_) : stackOfElt Int -> stackOfEltInt .
op _.q : stackOfEltInt -> Int .
op _.s : stackOfEltInt -> stackOfElt .

eq ((S .push(E)) .s) .top() .q = E .
...
endfm
```

We know class *stackOfElt* is inheriting from class *stack*. Our program will check for overridden every time when modelling each new method in class *stackOfElt*. If there is a method in class *stackOfElt* with the same name as one of the methods in class *stack*, we take class *stack's* method operators and tag "*base.*" between the "_." notation and the method name for each reference to the method name. So we have a new line in our Maude code for class *stackOfElt*:

```
op _.base.top() : stackOfElt -> stackOfEltInt .
```

Note that we retain the inheriting class' original return types[8]. And now we need to generate the equations for the overriding method:

```
var A : stackOfElt .
eq A .base.top().q = 0 .
```

The equations are simply copied from class *stack* and slightly modified by the tag "*base.*" as we did with the operators.

The example *stackOfElt* is just a simple example. It is possible we have several selected C# classes and they have more than one level in their inheritance tree, which we called multiple level inheritances. We will consider this case in later

[8] The behaviour of the inherited methods could be virtual or non-virtual. For example:
Stack x = new stackOfElt()
What is the class of x? There is 2 possibilities:
1. virtual, the class of x is stackOfElt
2. non-virtual, the class of x is stack
Java makes the determination at runtime of the type of the class of x is stackOfElt. But in C#, it could be either, however by default, it makes the determination at compile time and consider the class of x is stack.

chapters.

4.4 The Future Development of the Current System

The implementation of the current system is really just a proof of concept, rather than an attempt to produce a production system. The following list is the work which have not been implemented:

- A well designed XML format definition for "the constructor" of the attributes in EADF.
- A well desgined XML format definition for ESC
- An improved C# application of Attributes Generator, which will able to automatically generate a complete C# source code of Class Reader.

We will show a proposal of the solution to solve these problems in Chapter 6.

Chapter 5 Examples of Class Specification

In this chapter, we will test several selected simple C# classes to see if we can model them accurately using the algebraic specification attributes we have defined. These examples are taken from [1] and have been modified to fit our needs – for example changing Java to C#.

5.1 ArrayList

ArrayList is a commonly-used collection class in both C# and Java, with a range of operations. We define our own simple version of *MyArrayList* here. An ArrayList structure is defined in a similar way to the *stackOfElt* example. Note that a strong representation of ArrayList would include the ability deal with Exceptions. We omit that here. An ArrayList is defined as an empty ArrayList followed by add calls which add objects to the ArrayList. First we will examine the program of this *MyArrayList* class:

```
//an example of Arraylist
    [Hidden("op EmptyArrayList : -> MyArrayList")]
    public class MyArrayList
    {
        [Vars("C:Int;E:Int;A:MyArrayList")]
        public int Cap; //capicity of the arraylist
        public int Elt; //element of the arraylist
        public int Count;

        //constructor --- new. empty arraylist
        public MyArrayList()
        {...}

        //constructor -- new arraylist with a specified capcity
        public MyArrayList(int Cap)
        {...      }

        //method add, similar with push in Stack
        [Eq("(((A  .add(E))  .s)  .remove(E))==  A  .",MType="Command",Comment="A  is
ArrayList, E is Element.")]
        public void add (int Elt)
        {...      }
```

//method clear

[Eq("(A .clear())== EmptyArrayList .", MType="Command", Comment="A is ArrayList")]

public void clear()

{... }

//method size

[Eq("(((A .add(E)) .s) .size()) == (A .size()) + 1 .", Mtype="Query", Comment="A is ArrayList, E is Element")]

public int size()

{..}

//method contains

[Eq("(((A .add(E)) .s) .contains(E))== true .", MType = "Query", Comment = "A is ArrayList, E is Element .")]

public bool contains(int Elt)

{...}

//method insert, insert an element into a specified location of arraylist

//The way this works is that if the element to be added is really at the end anyway, we just use add;

*//Otherwise we move back through the list recursively (using the 2nd equation) until we get to a //point Where the first equation *does* apply*

[CEqs(" (A .insert(C,E)) .s == ((A .add(E)) .s) if A .size() = C . ; (((A .add(F)) .s) .insert(C.E)) .s == (((A .insert(C.E)) .s) .add(F)) .s) if ((A.add(F)).s) .size() =/= C .", MType="Query",Comment="A is ArrayList, E and F are Elements, C is location of the element.")]

public void insert (int Count, int Elt)

{... }

//method remove, remove the most recently added copy of an element

[Eq("(((A .add(E)) .s) .remove(E))== A .", MType = "Command", Comment= "A is ArrayList, E is Element .")]

public void remove(int Elt)

{... }

//method removeAt, remove an element at sepecified location

[Eq("(((A .insert(C,E)) .s) .removeAt(C))== A .",MType="Command",Comment="A is ArrayList, E is Element, C is location of the element.")]

public void removeAt (int Count)

{... }

//method indexOf, Returns the zero-based index of the first occurrence of a value in the ArrayList

[Eq("(((A .insert(C,E)) .s) .indexOf(E))== C .", MType ="Query", Comment="A is ArrayList, E is Element, C is location of the element.")]
public int indexOf(int Elt)
{... }

//method lastIndexOf, Returns the zero-based index of the last occurrence of a value in the ArrayList
[Eq("(((A .add(E)) .s) .removeAt((A .lastIndexOf(E)) .q))== A .", MType ="Command", Comment="A is ArrayList, E is Element")]
public int lastIndexOf(int Elt)
{... }
}

// method elementAt(int Count) that access the element at location C
[CEqs("((A .add(E)) .s) .elementAt(C) == E if ((A .add(E)) .s) .size() = C+1 . ; ((A .add(E)) .s) .elementAt(C) == A .elementAt(C) if ((A.add(E)) .s) .size() =/= C+1 .", MType="Query",Comment="A is ArrayList, E is Element, C is location of the element.")]
Public int elementAt(int Count)
{...}

The class is embedded with three attributes *Hidden, Vars* and *Eq*. These attributes have been already introduced in the above chapters. We are interested to see the GFS we would get from the *MyArrayList* class. Let us have a look at the methods of the *MyArrayList* class:

//method add, similar with push in Stack, Add one element into the ArrayList
public void add (int Elt)
{... }

//method clear, Delete all the elements in the ArrayList
public void clear()
{... }

//method size, the size of the ArrayList
public int size()
{..}

//method contains, Check if the ArrayList has the specified element
public bool contains(int Elt)
{... }

//method insert, Insert an element into a specified location of arraylist

```
public void insert (int Count, int Elt)
{... }

//method remove, Similar with pop in Stack, Remove an element of arraylist
public void remove(int Elt)
{... }

//method removeAt, advanced version of remove at specified location
public void removeAt (int Count)
{... }

//method indexOf, Returns the zero-based index of the first occurrence of a value in the ArrayList
public int indexOf(int Elt)
{... }

//method lastIndexOf , Returns the zero-based index of the last occurrence of a value in the ArrayList
public int lastIndexOf(int Elt)
{... }

//method elementAt, access the element at location C
Public int elementAt(int Count)
{...}
```

The above methods' list gives us a clear idea of their main functions and types. C# has its own integrated ArrayList class. We can find its members' list at [76], the methods we created here have the similar function to them. Now we use the Class Reader to generate the XML Class Specification:

```
?xml version="1.0"?>
<Class>
	<Name>MyArrayList</Name>
	<Extend />
	<Hidden>
		<Operation>op EmptyArrayList : -&gt; MyArrayList</Operation>
		<Comment />
	</Hidden>
	<Field Type="Int">C</Field>
	<Field Type="Int">E</Field>
	<Field Type="MyArrayList">A</Field>
	<Method Type="Command">
		<Name>add</Name>
		<ParameterType>System.Int32</ParameterType>
		<MethodReturnType>System.Void</MethodReturnType>
```

```
        <Equation>
            <LHS>(((A .add(E)) .s) .remove(E))</LHS>
            <RHS> A .</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>A is ArrayList, E is Element.</Comment>
    </Method>
    <Method Type="Command">
        <Name>clear</Name>
        <ParameterType />
        <MethodReturnType>System.Void</MethodReturnType>
        <Equation>
            <LHS>(A .clear())</LHS>
            <RHS> EmptyArrayList .</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>A is ArrayList</Comment>
    </Method>
    <Method Type="Query">
        <Name>size</Name>
        <ParameterType />
        <MethodReturnType>System.Int32</MethodReturnType>
        <Equation>
            <LHS>(((A .add(E)) .s) .size())</LHS>
            <RHS> (A .size()) + 1</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>A is ArrayList, E is Element .</Comment>
    </Method>
    <Method Type="Query">
        <Name>contains</Name>
        <ParameterType>System.Int32</ParameterType>
        <MethodReturnType>System.Boolean</MethodReturnType>
        <Equation>
            <LHS>(((A .add(E)) .s) .contains(E))</LHS>
            <RHS> true .</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>A is ArrayList, E is Element .</Comment>
    </Method>
    <Method Type="Query">
        <Name>insert</Name>
        <ParameterType>System.Int32System.Int32</ParameterType>
        <MethodReturnType>System.Void</MethodReturnType>
```

```
        <Equation>
            <LHS>(A .insert(C,E)) .s </LHS>
            <RHS>((A .add(E)) .s)</RHS>
        </Equation>
        <Condition> A .size() = C </Condition>
        <Equation>
            <LHS>(((A .add(F)) .s) .insert(C.E)) .s </LHS>
            <RHS> (((A .insert(C.E)) .s) .add(F)) .s)</RHS>
        </Equation>
        <Condition>((A.add(F)).s) .size() =/= C </Condition>
        <Comment> A is ArrayList, E and F are Elements, C is location of the element
        </Comment>
    </Method>
    <Method Type="Command">
        <Name>remove</Name>
        <ParameterType>System.Int32</ParameterType>
        <MethodReturnType>System. Void</MethodReturnType>
        <Equation>
            <LHS>(((A .add(E)) .s) .remove(E))</LHS>
            <RHS> A .</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>A is ArrayList, E is Element .</Comment>
    </Method>
    <Method Type="Command">
        <Name>removeAt</Name>
        <ParameterType>System.Int32</ParameterType>
        <MethodReturnType>System. Void</MethodReturnType>
        <Equation>
            <LHS>(((A .insert(C,E)) .s) .removeAt(C))</LHS>
            <RHS> A .</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>A is ArrayList, E is Element, C is location of the element.</Comment>
    </Method>
    <Method Type="Query">
        <Name>indexOf</Name>
        <ParameterType>System.Int32</ParameterType>
        <MethodReturnType>System.Int32</MethodReturnType>
        <Equation>
            <LHS>(((A .insert(C,E)) .s) .indexOf(E))</LHS>
            <RHS> C .</RHS>
        </Equation>
        <Condition>NULL</Condition>
```

```
            <Comment>A is ArrayList, E is Element, C is location of the element.</Comment>
        </Method>
        <Method Type="Command">
            <Name>lastIndexOf</Name>
            <ParameterType>System.Int32</ParameterType>
            <MethodReturnType>System.Int32</MethodReturnType>
            <Equation>
                <LHS>(((A .add(E)) .s) .removeAt((A .lastIndexOf(E)) .q))</LHS>
                <RHS> A .</RHS>
            </Equation>
            <Condition>NULL</Condition>
            <Comment>A is ArrayList, E is Element</Comment>
        </Method>
        <Method Type="Query">
            <Name> elementAt</Name>
            <ParameterType>System.Int32</ParameterType>
            <MethodReturnType> System.Int32</MethodReturnType>
            <Equation>
                <LHS>((A .add(E)) .s) .elementAt(C)</LHS>
                <RHS> E </RHS>
            </Equation>
            <Condition>((A .add(E)) .s) .size() = C+1</Condition>
            <Equation>
                <LHS>((A .add(E)) .s) .elementAt(C)</LHS>
                <RHS> A .elementAt(C)</RHS>
            </Equation>
            <Condition>((A.add(E)) .s) .size() =/= C+1</Condition>
            <Comment>A is ArrayList, E is Element, C is location of the element.</Comment>
        </Method>
</Class>
```

Reading the above XML Class specification is not a pleasure, and so we are transforming it into a generic Algebraic Specification by using the appropriate XSL stylesheet:

```
Class MyArrayList :: Specification {
Int: C
Int: E
MyArrayList: A
Hidden {
op EmptyArrayList : -> MyArrayList
//
}
Method add : System.Int32 -> System.Void [Command]{
```

```
eq (((A .add(E)) .s) .remove(E)) = A .
IF ( NULL )
//A is ArrayList, E is Element.
}
Method clear : -> System.Void [Command]{
eq (A .clear()) = EmptyArrayList .
IF ( NULL )
//A is ArrayList
}
Method size : -> System.Int32 [Query]{
eq (((A .add(E)) .s) .size()) == (A .size()) + 1 .
IF ( NULL )
//A is ArrayList, E is Element .
}
Method contains : System.Int32 -> System.Boolean [Query]{
eq (((A .add(E)) .s) .contains(E)) = true .
IF ( NULL )
//A is ArrayList, E is Element .
}
Method insert : System.Int32System.Int32 -> System.Void [Query]{
ceq  (A .insert(C,E)) .s == ((A .add(E)) .s) .
IF (A .size() = C)
ceq  (((A .add(F)) .s) .insert(C.E)) .s == (((A .insert(C.E)) .s) .add(F)) .s) .
IF (((A.add(F)).s) .size() =/= C)
// A is ArrayList, E and F are Elements, C is location of the element.
}
Method remove : System.Int32 -> System.Void [Command]{
eq (((A .add(E)) .s) .remove(E)) = A .
IF ( NULL )
//A is ArrayList, E is Element .
}
Method removeAt : System.Int32 -> System.Void [Command]{
eq (((A .insert(C,E)) .s) .removeAt(C)) = A .
IF ( NULL )
//A is ArrayList, E is Element, C is location of the element.
}
Method indexOf : System.Int32 -> System.Int32 [Query]{
eq (((A .insert(C,E)) .s) .indexOf(E)) = C .
IF ( NULL )
//A is ArrayList, E is Element, C is location of the element.
}
Method lastIndexOf : System.Int32 -> System.Int32 [Command]{
eq (((A .add(E)) .s) .removeAt((A .lastIndexOf(E)) .q)) = A .
IF ( NULL )
```

LIBRARY

```
//A is ArrayList, E is Element
}
Method elementAt : System.Int32 -> System.Int32 [Query]{
ceq  ((A .add(E)) .s) .elementAt(C) == E .
IF (((A .add(E)) .s) .size() = C+1)
ceq ((A .add(E)) .s) .elementAt(C) == A .elementAt(C) .
IF (((A.add(E)) .s) .size() =/= C+1)
// A is ArrayList, E is Element, C is location of the element.
}
}
```

The equations and types are clear in the example above, as is the relationship with the Maude code we can generate. By using the Translator, we can transform the *MyArrayList* class into a Maude Specification:

```
fmod MyArrayList is
protecting baseclass .
sort MyArrayList .
op EmptyArrayList : -> MyArrayList .

var C : Int .
var E : Int .
var A : MyArrayList .

op _.add(_) : MyArrayList Int -> MyArrayListVoid .

sort MyArrayListVoid .
op (_,_) : MyArrayList Void -> MyArrayListVoid .
op _.q : MyArrayListVoid -> Void .
op _.s : MyArrayListVoid -> MyArrayList .

op _.clear() : MyArrayList -> MyArrayListVoid .
op _.size(): MyArrayList -> MyArrayListInt .

sort MyArrayListInt .
op (_,_) : MyArrayList Int -> MyArrayListInt .
op _.q : MyArrayListInt -> Int .
op _.s : MyArrayListInt -> MyArrayList .

op _.contains(_) : MyArrayList Int -> MyArrayListBool .

sort MyArrayListBool .
op (_,_) : MyArrayList Bool -> MyArrayListBool .
op _.q : MyArrayListBool -> Bool .
```

```
op _.s : MyArrayListBool -> MyArrayList .

op _.insert(_,_) : MyArrayList Int Int -> MyArrayListVoid .
op _.remove(_) : MyArrayList Int -> MyArrayListVoid .
op _.removeAt(_) : MyArrayList Int -> MyArrayListVoid .
op _.indexOf(_) : MyArrayList Int -> MyArrayListInt .
op _.lastIndexOf(_) : MyArrayList Int -> MyArrayListInt .
op _.elementAt(_) : MyArrayList Int -> MyarrayListInt .

eq (((A .add(E)) .s) .remove(E)) .s = A .
eq (A .clear()) .s = EmptyArrayList .
eq (((A .add(E)) .s) .size()) == (A .size()) + 1 .
eq (((A .add(E)) .s) .contains(E)) .q = true .
ceq (A .insert(C,E)) .s == ((A .add(E)) .s) if A .size() = C .
ceq (((A .add(F)) .s) .insert(C.E)) .s == (((A .insert(C.E)) .s) .add(F)) .s) if ((A.add(F)).s) .size()
=/= C .
eq (((A .add(E)) .s) .removeAt((A .lastIndexOf(E)) .q)) .s = A .
ceq ((A .add(E)) .s) .elementAt(C) == E if ((A .add(E)) .s) .size() = C+1 .
ceq ((A .add(E)) .s) .elementAt(C) == A .elementAt(C) if ((A.add(E)) .s) .size() =/= C+1 .
endfm
```

Observe that we make use of the sort tupling described earlier (Section 4.3.5), together with tupling and projection operators. E.g. *"MyArrayListInt"," op (_,_) : MyArrayList Int -> MyArrayListInt"*, and *"_.q : MyArrayListInt -> Int (and so on)"*. In this particular example, this may be considered unnecessary because there are no methods that are both commands and queries. However, this code is automatically generated.

Currently we can not guarantee all the C# system types can be normalized into Maude types (simply because not all types in C# have a corresponding type in Maude) and we can not handle the operations which have a lot of parameters.

5.2 The Shapes Example

In this section we will show the examples of *Shape, Rectangle and Square*[9] as a class specification and the full Maude specification that will be produced from them. Also we consider inheritance again since the shape example includes multiple levels. The importing of methods from an inherited class is reasonably trivial but the modelling of method overriding and accessing overridden

[9] These examples are taken from [1] and have been rewritten into C# version. In [1], all of the examples are based on Java.

methods is more complex. In our examples and in C # generally, there exists only single inheritance so our model does not have to cope with multiple inheritance. However, we still have to be able to model a class with many *levels* of inheritance if needed.

For example, if we 3 classes in C++: *class washingdryer, class washingmachine, class tumbledryer*, we have the following mutilple inheritances:

class washingdryer : washingmachine, tumbledryer

It can be represented in Maude as:

```
sort washingdryer .
subsort washingdryer < washingmachine .
subsort washingdryer < tubledryer .
```

Class *Shape* is a general class for geometric shapes. It can be assumed that all classes that inherit from this class will have the general attributes of *Shape*. We know it is not a good way to write the Shape C# programs as we did, because we do not really need to override the area method. But we are doing it to show how overriding works when we model it in Maude.

The following is a very simple version of Class *Shape*:

```
//an example of Shape
[Hidden("op AShape : -> Shape")]
public class Shape
{
    [Vars("S:Shape")]
    public Shape S;     // see below

    //constructor
    public Shape()
    {}

    [Eq("S.area() == 0",MType="Query",Comment="S is Shape")]
    public int area()
    {
        return 0;
    }
}
```

Note that we are using the public field *S* purely as a placeholder to locate the *[Vars]* attribute. Attributes in C# must be associated with some element(s):

fields, methods, classes etc. This is somewhat annoying and we hope that a future version of the software would eliminate this need.

The corresponding generated XML Class Specification is:

```
<?xml version="1.0"?>
<Class>
	<Name>Shape</Name>
	<Extend/>
	<Hidden>
		<Operation>op AShape : -&gt; Shape</Operation>
		<Comment />
	</Hidden>
	<Field Type="Shape">S</Field>
	<Method Type="Query">
		<Name>area</Name>
		<ParameterType />
		<MethodReturnType>System.Int32</MethodReturnType>
		<Equation>
			<LHS>S.area()</LHS>
			<RHS>0</RHS>
		</Equation>
		<Condition>NULL</Condition>
		<Comment>S is Shape</Comment>
	</Method>
</Class>
```

After applying the XSL transformation we get the following:

```
Class Shape Specification {
Shape: S
Hidden {
op AShape : -> Shape
//
}
Method area : -> System.Int32 [Query]{
eq S.area() = 0 .
IF ( NULL )
//S is Shape
}
}
```

Class *Rectangle* will inherit from *Shape*. However *Rectangle* is a more specific Shape and is therefore less general. This means that it will override the Shape's

method area with Rectangle's own definition. Bellow are the C# code and XML class specification for *Rectangle*:

```
//an example of Rectangle
[Hidden("op ARectangle : -> Rectangle")]
public class Rectangle:Shape
{
    [Vars("R:Rectangle ; Side1:int; Side2:int")]
    public Rectangle R;
    private int Side1;
    private int Side2;

    //constructor
    public Rectangle()
    {}

[Eq("R.setSide1(Side1).setSide2(Side2).area() == Side1*Side2",
MType="Command",Comment="R is Rectangle")]
public void setSide1( int side1)
    {
        Side1 = side1;
    }

[Eq("R.setSide1(Side1).setSide2(Side2).area() == Side1*Side2",
MType="Command",Comment="R is Rectangle")]
public void setSide2( int side2)
    {
        Side2 = side2;
    }

[Eq("R.setSide1(Side1).setSide2(Side2).area() == Side1*Side2",
MType="Command",Comment="R is Rectangle")]
    public int area()
    {
        return Side1*Side2;
    }
}
```

And the corresponding generated XML Class Specification is:

```
<?xml version="1.0"?>
<Class>
    <Name>Rectangle</Name>
    <Extend>Shape</Extend>
```

```
        <Hidden>
                <Operation>op ARectangle : -&gt; Rectangle</Operation>
                <Comment />
        </Hidden>
        <Field Type="Rectangle">R</Field>
        <Field Type="int"> Side1</Field>
        <Field Type="int"> Side2</Field>
        <Method Type="Command">
                <Name>setSide1</Name>
                <ParameterType>System.Int32</ParameterType>
                <MethodReturnType>System.Void</MethodReturnType>
                <Equation>
                        <LHS>R.setSide1(Side1).setSide2(Side2).area()</LHS>
                        <RHS>Side1*Side2</RHS>
                </Equation>
                <Condition>NULL</Condition>
                <Comment>R is Rectangle</Comment>
        </Method>
        <Method Type="Command">
                <Name>setSide2</Name>
                <ParameterType>System.Int32</ParameterType>
                <MethodReturnType>System.Void</MethodReturnType>
                <Equation>
                        <LHS>R.setSide1(Side1).setSide2(Side2).area()</LHS>
                        <RHS>Side1*Side2</RHS>
                </Equation>
                <Condition>NULL</Condition>
                <Comment>R is Rectangle</Comment>
        </Method>
        <Method Type="Command">
                <Name>area</Name>
                <ParameterType />
                <MethodReturnType>System.Int32</MethodReturnType>
                <Equation>
                        <LHS>R.setSide1(Side1).setSide2(Side2).area()</LHS>
                        <RHS>Side1*Side2</RHS>
                </Equation>
                <Condition>NULL</Condition>
                <Comment>R is Rectangle</Comment>
        </Method>
</Class>
```

Then we get:

```
Class Rectangle ::Shape Specification {
Rectangle: R
int: Side1
int: Side2
Hidden {
op ARectangle : -> Rectangle
//
}
Method setSide1 : System.Int32 -> System.Void [Command]{
eq R.setSide1(Side1).setSide2(Side2).area() = Side1*Side2 .
IF ( NULL )
//R is Rectangle
}
Method setSide2 : System.Int32 -> System.Void [Command]{
eq R.setSide1(Side1).setSide2(Side2).area() = Side1*Side2 .
IF ( NULL )
//R is Rectangle
}
Method area : -> System.Int32 [Command]{
eq R.setSide1(Side1).setSide2(Side2).area() = Side1*Side2 .
IF ( NULL )
//R is Rectangle
}
}
```

The final class we consider is *Square*. This will inherit from *Rectangle*. This means that *Square* is not only a more specialised form of *Shape* but also a more specialized form of *Rectangle*. This means it will override *Rectangle*'s method. The following are the C# code and XML class specification for *Square*:

```
//an example of Square
    [Hidden("op ASquare : -> Square")]
    //[Hidden("op Side : -> Int")]
    public class Square:Rectangle
    {
        [Vars("S:Square;Side:int")]
        public Square S;
        private int Side;

        //constructor
        public Square()
        {}

        [Eq("S.setSide(Side).area()    ==    Side*Side",MType="Command",Comment="S    is
```

```
Square")]
        public void setSide( int side)
        {
            Side = side;
        }

        //method add, similar with push in Stack
        [Eq("S.setSide(Side).area()    ==    Side*Side",MType="Command",Comment="S    is
Square")]
        public int area()
        {
            return Side*Side;
        }
    }
```

And the corresponding generated XML Class Specification is:

```
<?xml version="1.0"?>
<Class>
    <Name>Square</Name>
    <Extend>Rectangle</Extend>
    <Hidden>
        <Operation>op ASquare : -&gt; Square</Operation>
        <Comment />
    </Hidden>
    <Field Type="Square">S</Field>
    <Field Type="int"> Side</Field>
    <Method Type="Command">
        <Name>setSide</Name>
        <ParameterType>System.Int32</ParameterType>
        <MethodReturnType>System.Void</MethodReturnType>
        <Equation>
            <LHS>S.setSide(Side).area()</LHS>
            <RHS>Side*Side</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>S is Square</Comment>
    </Method>
    <Method Type="Command">
        <Name>area</Name>
        <ParameterType />
        <MethodReturnType>System.Int32</MethodReturnType>
        <Equation>
            <LHS>S.setSide(Side).area()</LHS>
```

```
            <RHS>Side*Side</RHS>
        </Equation>
        <Condition>NULL</Condition>
        <Comment>S is Square</Comment>
    </Method>
</Class>
```

Then we get:

```
Class Square :: Rectangle Specification {
Square: S
int: Side
Hidden {
op ASquare : -> Square
//
}
Method setSide : System.Int32 -> System.Void [Command]{
eq S.setSide(Side).area() = Side*Side
IF ( NULL )
//S is Square
}
Method area : -> System.Int32 [Command]{
eq S.setSide(Side).area() = Side*Side
IF ( NULL )
//S is Square
}
}
```

The methods which *Rectangle* inherits from *Shape* and *Square* inherits from *Rectangle* do not appear on the above XML Class Specification. But when we generated them into Maude Specifications they will be presented.

```
fmod Shape is
sort Shape .
op AShape : -> Shape .

var S : Shape .

op _.area() : Shape -> ShapeInt .

sort ShapeInt .
op (_,_) : Shape Int -> ShapeInt .
op _.q : ShapeInt -> Int .
op _.s : ShapeInt -> Shape .
```

```
eq S.area() .q =0
endfm
```

The Maude Specification of *Shape* has one operation and one equation, which will have an effect on the inheritance.

```
fmod Rectangle is
protecting Shape .
sort Rectangle .
subsort Rectangle < Shape .
op ARectangle : -> Rectangle .

var R : Rectangle .
var   Side1 : int .
var   Side2 : int .

op _.setSide1(_) : Rectangle Int -> RectangleVoid .

sort RectangleVoid .
op (_,_) : Rectangle Void -> RectangleVoid .
op _.q : RectangleVoid -> Void .
op _.s : RectangleVoid -> Rectangle .

op _.setSide2(_) : Rectangle Int -> RectangleVoid .
op _.area() : Rectangle -> RectangleInt .

subsort RectangleInt < ShapeInt .

op _.base.area() : Rectangle -> RectangleInt .
var S : Rectangle .
eq S.base.area() .q =0

sort RectangleInt .
op (_,_) : Rectangle Int -> RectangleInt .
op _.q : RectangleInt -> Int .
op _.s : RectangleInt -> Rectangle .


eq R.setSide1(Side1).setSide2(Side2).area() .q =Side1*Side2
endfm
```

Rectangle inherits from *Shape*, therefore in Maude Specification, it is a subsort

of *Shape*. And it overrides the operator *"area()"* from *Shape* which means we must introduce a new operator *"base.area()"*.

```
fmod Square is
protecting Rectangle .
sort Square .
subsort Square < Rectangle .
op ASquare : -> Square .

var S : Square .
var Side : Int .

op _.setSide(_) : Square Int -> SquareVoid .

subsort SquareVoid < RectangleVoid .

sort SquareVoid .
op (_,_) : Square Void -> SquareVoid .
op _.q : SquareVoid -> Void .
op _.s : SquareVoid -> Square .

op _.area() : Square -> SquareInt .

op _.base.area() : Square -> SquareInt .
var R : Square .
eq ((((R .setSide1(Side1)) .s) .setSide2(Side2)) .s) .base.area() .q = Side1 * Side2 .

subsort SquareInt < RectangleInt .

sort SquareInt .
op (_,_) : Square Int -> SquareInt .
op _.q : SquareInt -> Int .
op _.s : SquareInt -> Square .


eq ((S .setSide(Side)) .s) .area() .q = Side * Side .
endfm
```

Similar to *Rectangle*, there are additional equations and declarations generated in *Square* to allow us access to the super or base class' original methods and fields. This is particularly important if they have been overridden by the inheriting class *Rectangle*. However, there will be a warning message when testing *Square* in Maude regarding of equation

"*((((R .setSide1(Side1)) .s) .setSide2(Side2)) .s) .base.area() .q = Side1 * Side2*", because it refers to variables *Side1* and *Side2* which are defined in *Rectangle*, and not *Square*. Note that although this warning is irritating, it does not stop the Maude code working. It is possible to fix the warning message in two ways– either in the form of a "quick fix", by simply declaring the two variables *Side1* and *Side2* in the Attribute *Hidden*; or with "a long term fix", by creating a utility in our program to scan through the base class and parse the inheriting equations to find out the undeclared variables. This extended work can be considered as Type Inference - Hindley-Milner Algorithm [84], which we will leave here as it would be time-consuming to implement and would not influence this project a lot.

Chapter 6 Future Development

This Chapter will discuss plans for future development and possible expansion. We will discuss how to model a C# class in a universal data format which could be adapted to many other object-oriented languages. We will explain the modelling process of generating a Pre-Post condition formal specification by using our current modelling system. We will also discuss the possible solutions for those unsolved problems in the development of this project.

6.1 A Prospect for a Universal Structure

The long-term aim of the work described in this thesis is to create a generic software framework that (a) can be adapted to a range of object oriented languages and (b) enables a wide range of formal methods to be embedded as documentation/specification within program code. So far, the framework only works for a single language, and has been demonstrated for only one documentation/specification formalism. Further work is needed to permit other languages to be used, and to demonstrate that the framework can be used with other formalisms. We begin to address the second point In Section 6.3, where the application of the current framework to a pre/post-condition documentation/specification model is considered. Below, we outline consider how the framework can be adapted to accommodate other languages.

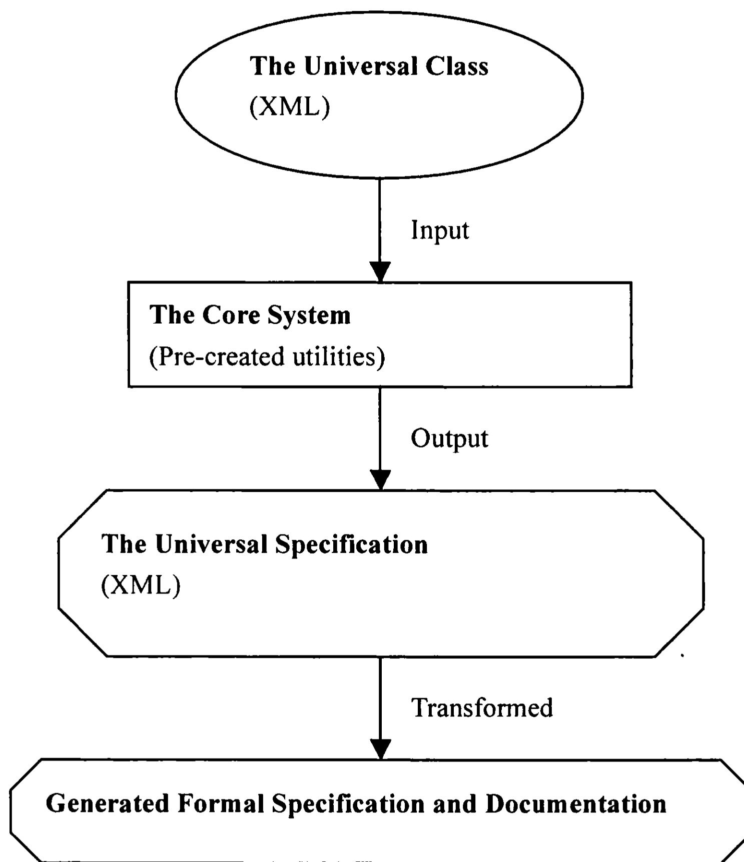


Figure6_1: The Universal Structure

In Figure6_1, the universal class is an object-oriented class in the XML format. We can generate the universal specification from the universal class, which is also in the XML format, and then we can use XML stylesheet to transform the universal specification into some other formal specification and documentation format. We propose a modification of our current system with the aim of moving towards a universal structure:

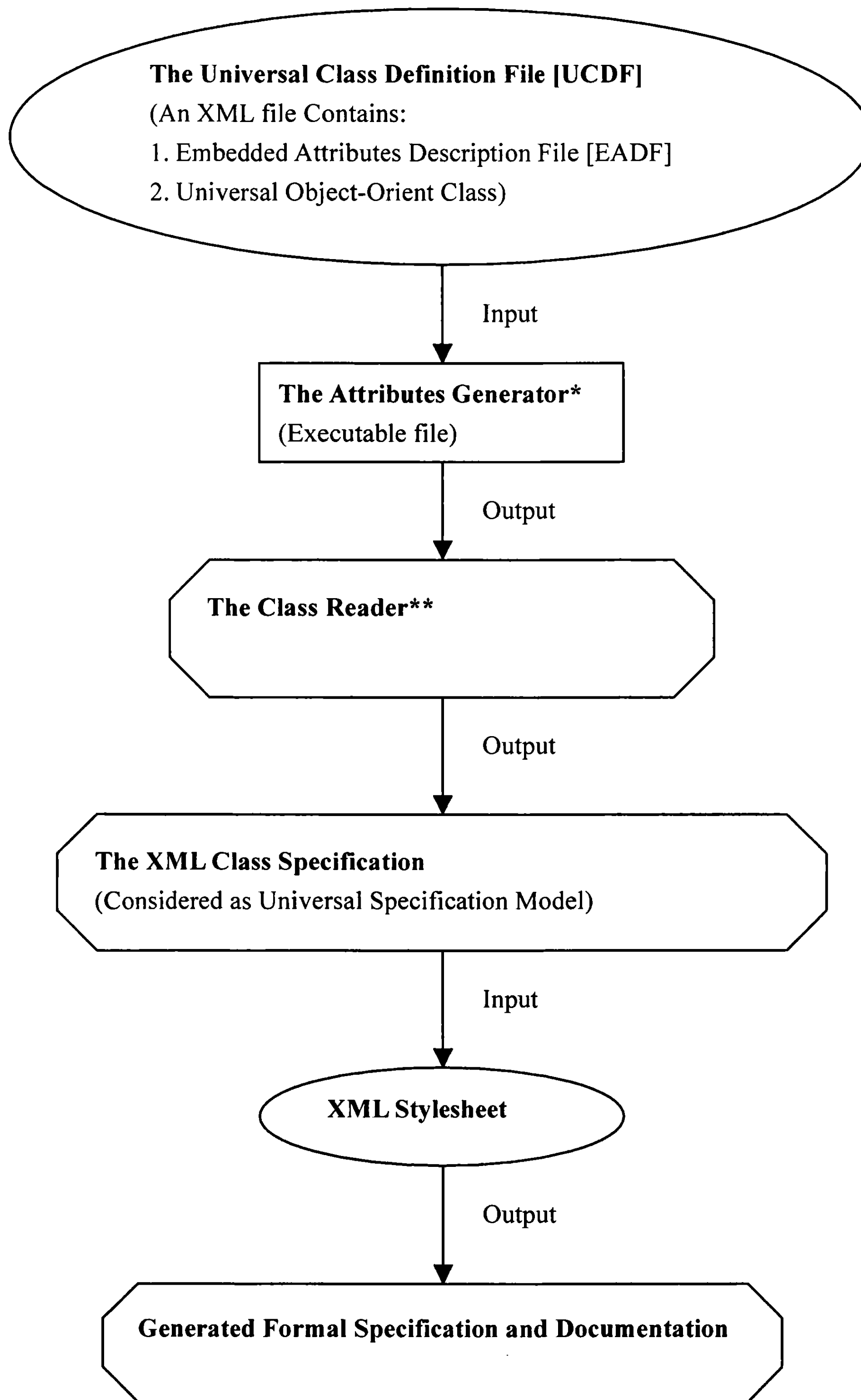


Figure6_2: New expanded System

Compared with the system we have described so far (Figure3_5), some significant changes have been proposed. We will explain each part which is different from the corresponding part in Figure3_5:

The Universal Class Definition File [UCDF]

The Universal Class Definition File is an XML file which contains two parts: the Embedded Attributes Description File [EADF] and the Universal Object-Orient Class (see section 6.2). Here we want to create a universal model for an Object-Oriented Class that can be adapted to a range of object-oriented language and not only C#. Obviously, XML is a good representation for this universal model in this project. We combine the EADF with this universal object-orient class into the UCDF.

The Attributes Generator*

The attributes generator here is slightly different from the one in Chapter 4. It also needs to read information of the object-oriented class embedded in the UCDF and generate the class in C# class format. Now we can consider this generator as a "ClassReader and ESC generator".

The XML Stylesheet

The XML Stylesheet is considered as an important utility to transform the XML class specification into various formats.

The Generated Formal Specification and Documentation

This could be any specification and documentation depending on the XML Stylesheet.

6.2 A Prototype of the Universal Object-Orient Class Model

In this section we will explain how we could define the universal Object-orient class in XML. Let us look at the example code of the universal model in the UCDF (the code only contains the general structure of the universal model in XML format):

```
<Compound_Attribute_Class>
        <Compound_Attribute>
            <C_Name>Hidden</C_Name>
            <C_Attribute>Input anything you like</C_Attribute>
        </Compound_Attribute>
        <Name>Temp_class</Name>
        <Compound_Attribute>
            <C_Name>Vars</C_Name>
            <C_Attribute>Input types of variables here, ie: E:int</C_Attribute>
        </Compound_Attribute>
```

```
        <Constructor>
            public int Elt;
            (type your own vars. here)
        </Constructer>
        <Compound_Attribute>
            <C_Name>Eq</C_Name>
            <C_Attribute>Input equations here, ie:x=y</C_Attribute>
        </Compound_Attribute>
        <Compound_Attribute_Function>
            <C_Name>pop</C_Name>
            <C_Type>void</C_Type>
            <Constructor>
                if (myStack.Count>0)
                        myStack.Pop();
                //type anything here that should be in your function
                //u may have more than one functions
            </Constructor>
         </Compound_Attribute_Function>
   </Compound_Attribute_Class>
```

Example6_1: the Universal Model in XML format

The tag *<Compound_Attribute_Class>...</Compound_Attribute_Class>* encloses everything needed to generate the ESC. The tag *<Compound_Attribute>...</Compound_Attribute>* contains the data required for the embedded attributes of the Example Class. It normally has 2 elements: *<C_name>* and *<C_Attribute>*. Tag *<C_Name>* encloses the name of the embedded attribute and the tag *<C_Attributes>* encloses the proper content of the embedded attributes.

```
[Hidden("op EmptyStack: -> stackOfElt")]
public class stackOfElt
{...  }
```

For example, in order to generate the above C# code, then we should have the following code in EADF (obviously we are using the same structure as in Example6_1):

```
<Compound_Attribute_Class>
    <Compound_Attribute>
            <C_Name>Hidden</C_Name>
            <C_Attribute> op EmptyStack: -> stackOfElt </C_Attribute>
    </Compound_Attribute>
    <Name>stackOfElt</Name>
  ...
```

```
</Compound_Attribute_Class>
```

Clearly the tag *<Name>...</Name>* encloses the name of the ESC. And the element *<Constructor>* is similar to the one we introduced before (see Chapter 4.1.1).

At this stage we have not done anything special about the content enclosed in element *<C_Attribute>*, however, we purpose to format the content in a general regular expression-based structure. For example, if we have:

```
<C_Name>Eq</C_Name>
<C_Attribute> "a=b", "e=f", "g=h", "a big test" </C_Attribute>
```

We could have the following generated code:

```
[Eq("a=b", "a big test")]
[Eq("e=f", "a big test")]
[Eq("g=h", "a big test")]
public void testcode()
{...  }
```

The tag *<Compound_Attribute_Function>...</Compound_Attribute_Fuction>* encloses full details of a method of the selected C# Class, which has 3 different elements: *<C_Name>, <C_Type>* and *<Constructer>*.

```
public void pop()
{
    if(myStack.Count>0)
        myStack.Pop();
}
```

For the above example method, we should have the following code in XML:

```
<Compound_Attribute_Function>
    <C_Name>pop</C_Name>
    <C_Type>void</C_Type>
    <Constructer>
        if (myStack.Count>0)
            myStack.Pop();
    </Constructer>
 </Compound_Attribute_Function>
```

As before *<C_Name>...</C_Name>* encloses the name of the method and *<C_Type>...</C_Type>* encloses the type of the method, and the element

<*Constructor*> is also as before. The following is generated from Example6_1:

```
[Hidden(Input anything you like)]
public class Temp_class
{
[Vars(Input types of variables here, ie: E:int)]

        public int Elt;
        (type your own vars. here)

[Eq(Input equations here, ie:x=y)]
public void pop()
{

            if (myStack.Count>0)
                myStack.Pop();
            //type anything here that should be in your function
            //u may have more than one functions

}
}
```

Since we are now considering the model to be "universal", it should be easy to adapt to at least some other object oriented programming languages. The following is a simple Java class without attributes:

```
class ASimpleJava {

    int number = 0;

    void printStates() {
        System.out.println("number:"+number);
    }
}
```

Example6_2: A Simple Java Class

The Java in Example6_2 can be transformed into the following:

```
<Compound_Attribute_Class>
    <Name>ASimpleJava</Name>
    <Constructor>
        Public int number = 0;
    </Constructor>
    <Compound_Attribute_Function>
```

```
        <C_Name>printStates</C_Name>
        <C_Type>void</C_Type>
        <Constructor>
            System.write("cadence:"+cadence);
        </Constructor>
    </Compound_Attribute_Function>
</Compound_Attribute_Class>
```

Example6_3: The Java Class in the Universal Model

Note that although Java, like C#, has reflection capabilities it has historically lacked a counterpart to attributes. The usual mechanism for embedding meta information is to use an extension of the javadoc mechanism of special comments [1] [81]. However, newer versions of Java include *annotations*, which provide similar functionality to attributes. We do not consider these further here.

It should be noted that the universal model is still in a very early version and incomplete. It is still lacks many features to represent a complicated object oriented class, for example: if a method has parameters (i.e. *push(int ANumber)*), we do not have any element defined for them in the current model. Also the element *<Constructor>* here works like a string tag. And it could be noticed in Example6_3, the code in the tag *<Constructor>* has been modified into C# style manually. There are many issues with the currently-proposed universal model, but they can be fixed in the future development. The intention of this section is to show the flexibility of our original system and how we could model the universal object-orient class in times to come.

6.3 An Alternative Formalism: Pre-Post Conditions

In this section, we will discuss how to generate formal specifications and documentation based on Pre-Post conditions. Pre and Post-Conditions are typically embedded within programs (or specifications) to formally define program semantics (in other words, to document it). A Pre-Condition specifies conditions that must apply before a code fragment can successfully execute, and a Post-Condition specifies what can be assumed to be true after successful execution. Pre and Post-Conditions do not in themselves constitute a formal specification paradigm. However, they form an important part of formalisms such as Hoare Logic (also called Floyd-Hoare Logic) [85]. They are also important in models based on Weakest Pre-Conditions [86], [87]. A commonly-related concept is the invariant – that is used to define the behaviour

of loops. However, since our model is only concerned with the behaviour of programs at the level of public class components, and not lower-level constructs (loops, conditionals etc.), invariants are not something we can usefully represent.

A variety of notations are used to represent Pre and Post-Conditions. The most commonly-seen is the Hoare Triple:

$\{P\}\ Q\ \{R\}$

Meaning given Pre-Condition *P*, execution of program fragment *Q* will result in Post-Condition *R* being true.

Since we have developed a technique that is intended to be generic, this is an 'experiment' to show the potential flexibility of our developed system that it applies to formal methods other than just algebraic specifications. Consequently, this example is just shown as a proof of principle, and not developed in depth. We will need to create a two new attributes for representing a Pre-condition and a Post-condition:

```
[Pre("condition")]
[Post("condition")]
```

The type of the parameter condition is string. Now we add these two attributes into the C# class Square from Chapter 6:

```
//an example of Shape
 [Pre("x >= 0")]
 [Post("sqrt(x) * sqrt(x) = x")]
 public class Square
 {
  public int Side;

  //constructor
  public Square()
  {}

  //method add, similar with push in Stack
  public int area()
  {
   return Side*Side;
  }
 }
```

After we input this Square class into the class reader, we would have the following XML class specification:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="PrePostSpec.xsl"?>
<Class>
    <Name>Square</Name>
    <Pre>
        <PreCondition>x &gt;= 0</PreCondition>
        <Comment />
    </Pre>
    <Post>
        <PostCondition>sqrt(x) * sqrt(x) = x</PostCondition>
        <Comment />
    </Post>
</Class>
```

It is easy to notice that we have used a new simple XML stylesheet *PrePostSpec.xsl* in the above class specification. The stylesheet will transform the universal class specification into the Pre-Post-condition specification:

```
[Pre-condition :: x >= 0 // ]
Class Square ::
[ Post-condition :: sqrt(x) * sqrt(x) = x // ]
Specification {
}
```

It is a very simple experiment and we have omitted much code and many other features to keep this example simple and clear. As we can see, with a complete and functional system based on the universal structure, there is little work needed to do to generate a specific formal specification and documentation as required.

6.4 Further Work

The following significant issues remain to be addressed:

- A consistent and complete Universal Class Model
- A fully generated and functional Class Reader (from Attributes Generator)
- A more friendly and easy to configurable and expandable XML stylesheet
- Good quality user tools

Refer to Chapter 6.1, there is still much work left to finish the Universal Class Model. It still requires that users have some basic knowledge of C# when they transform the object-oriented class into this model manually. Also it shares a weakness with the EADF (see Chapter 4.1.1), the content of tag <Constructor> contains strings that is not properly formatted and will be copied straight into generated C# files, which may cause unpredictable programming errors.

At the moment, the Class Reader could not be fully generated due to the difficulties of generating the main-code (see Chapter 4.1.2). The problem we mainly have is we are not sure of the targeted attributes location in the targeted ESC. The solution for this could be, to extend attributes to include location information, for example:

[Hidden("equations","comment","Location:Class")]
[Eq("equations","comment", "Location: Method")].

It is also possible to generate or automatically modify the XML stylesheet according to the UCDF. Currently, it is a limitation for us that we need to write a new XML stylesheet for every new specific formal specification style.

Also we could create a friendly and robust user interface for this system, it will be easy to run each utility and check the status of the system from the system log.

6.5 Conclusion

This thesis investigates the construction of a generic software framework which can provide a formal specification and documentation model for C# classes including important concepts such as inheritance. We have outlined a methodology that will allow other programmers to easily be able to declaratively define, using XML, the specifications and documentations for their own C# class. We have partially defined the declarative XML framework, and, specifically for C#, we have partially implemented the software system to translate XML descriptions of specification/documentation styles into operating, embeddable code. For those parts of the XML framework that are not yet defined and implemented, we have supplied a temporary 'bridge' in terms of example-specific code for the algebraic case. We have shown how, in the algebraic case, the generation of many of the equations defining the functionality of C# classes can be automated thus reducing the work in specifying new classes. We have shown how we have algebraically specified the functionality of C# classes using Maude.

The implementation had been done as following:

1. Allow User to customize "attributes" in the EADF.
2. Created a C# program (the Attributes Generator) which is able to transform the EADF (XML format) into C# source code.
3. Created a C# program (the Class Reader) which is able to extract the attributes from the ESC into the XML Class Specification.
4. Created a C# program (the translator) which is able to transform the XML Class Specification into the Maude Specification.
5. Proposed a prototype of a universal class model in XML format for Object oriented languages.

We would like to simplify the specification process as much as possible as the simpler the process is the easier it is for a user to specify classes. The framework and implementation are not yet complete (though this is to be expected, given the size of the project being attempted in the available timeframe). At present our framework specifically works with C#. With many object-oriented languages in Common use such as Java and C++ it would be desirable to adapt our framework to work with other languages. We feel that although our model is specifically aimed at C# it is sufficiently general in places when it refers to object-oriented concepts and features that it would be adapted to other object-oriented languages without a radical rewrite of the model of the modelling process.

Chapter 7 Bibliography

Most internet link sources are followed by an access date.

[1]:
Algebraically Modelling Java Object-Oriented Programs, Justin Biddle PH.D. Thesis, University of Wales, Swansea, 2005

[2]
http://msdn2.microsoft.com/en-us/library/tkxs89c5(vs.71).aspx, MSDN, *Adding Comments in C# Code, 17/05/2007*

[3]:
http://msdn2.microsoft.com/en-us/library/aa730781(vs.71).aspx, MSDN, *Viewing Code Structure with Comments, 17/05/2007*

[4]
http://java.sun.com/j2se/1.5.0/docs/api/, SUN official web site, *the API specification for the Java 2 Platform Standard Edition 5.0, 17/05/2007*

[5]
Design Goals of ACL2, Matt Kaufmann and J Moore, CLI Technical Report 101, Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, TX 78703, 1994

[6]
PVS : A Prototype Verification System, Sam Owre, Natarajan Shankar, and John Rushby, from CADE 11, Saratoga Springs, NY, June 1992.

[7]
The C++ programming language, Bjarne Stroustrup, Addison-Wesley, 2nd edition, 1991

[8]
The annotated C++ reference manual, Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, 1990

[9]

The UNIX C shell field guide, Gail Anderson and Paul Anderson, Prentice-Hall, 1986

[10]
The C Programming Language (2nd Edition), by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1988

[11]
http://java.sun.com/, Sun Microsystems official web site, 17/05/2007

[12]
http://java.sun.com/developer/technicalArticles/Security/whitepaper/JS_White_Paper.pdf, Java Security Overview, white paper, April 2005

[13]
http://java.sun.com/products/ejb/pdf/white_paper.pdf, White Paper, Sun Microsystems, 2000

[14]
The C# Programming Language, Anders Hejlsberg, Scott Wiltamuth and Peter Golde, Addison Wesley, 2003

[15]
Microsoft Visual C# .NET, Mickey Williams, Microsoft Press, 2002

[16]
Inside Visual Studio .NET Microsoft Press International, by Brian Johnson, Craig Skibo, and Marc Young, 2003

[17]
Programming C#, Jesse Liberty, O'Reilly & Associates, 2003

[18]:
http://www.ondotnet.com/pub/a/dotnet/2005/10/03/what-is-csharp.html?page=2, What is C#, by Jesse Liberty, 2005

[19]
http://msdn2.microsoft.com/en-us/library/byxd99hx(vs.71).aspx, MSDN, Using the WebMethod Attribute. 17/05/2007

[20]
http://msdn2.microsoft.com/en-us/library/xtwkdas5.aspx, MSDN, Attributes Overview, 17/05/2007

[21]
http://msdn2.microsoft.com/en-us/library/z0w1kczw.aspx, MSDN, C# Programming Guild: Attribute, 17/05/2007

[22]
http://msdn2.microsoft.com/en-us/library/sw480ze8.aspx, MSDN, Creating Custom Attributes, 17/05/2007

[23]
http://msdn2.microsoft.com/en-us/library/tw5zxet9.aspx, MSDN, AttributeUsage, 17/05/2007

[24]
http://msdn2.microsoft.com/en-us/library/ms173183.aspx, MSDN, C# Programming Guild: Reflection, 17/05/2007

[25]
http://msdn2.microsoft.com/en-us/library/t0cs7xez.aspx, MSDN, View Type Information, 17/05/2007

[26]
Inside C#, Tom Archer and Andrew Whitechapel, 2nd edition, Microsoft Press, 2002

[27]
http://msdn2.microsoft.com/en-us/library/609yztkt.aspx, MSDN, Reflection and Generic Types, 17/05/2007

[28]
http://msdn2.microsoft.com/en-us/library/f7ykdhsy.aspx, MSDN, Reflection Overview, 17/05/2007

[29]
http://msdn2.microsoft.com/en-us/library/z919e8tw.aspx, MSDN, Accessing Attributes with Reflection, 17/05/2007

[30]
http://www.w3.org/XML/, World Wide Web Consortium XML official web site

[31]
XAML in a Nutshell, Lori A. MacVittie, O'Reilly & Associates, Inc, 2006

[32]
An Exercise in Algebraic Specification with Total Functions, JA Bergstra, JV

Tucker, Programming Research Group Report, University of Amsterdam, 1988

[33]
http://www.w3.org/TR/2006/REC-xml-20060816/#syntax, specification for Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C, 2006

[35]
http://www.w3.org/Style/XSL/, W3C, XSL official web site, 17/05/2007

[36]
XSLT, Doug Tidwell, O'Reilly publishing, 2001

[37]
A Theory of Software Interfaces, D.Ll. L. RES, PH.D. thesis, University of Wales, Swansea, 2001

[38]
An Algebraic Approach to Syntax, Semantics and Compilation, Karen Stephenson, PH.D. Thesis, University of Wales, Swansea, 1996

[39]
Algebraic Models of Correctness for Abstract Pipelines, A.C.J. Fox and N.A. Harman, The Journal of Algebraic and Logic Programming, 57 (2003), 71-107.

[40]
The Maude 2.0 System. In Proc. Rewriting Techniques and Applications, Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer and Carolyn Talcott Springer-Verlag LNCS 2706, 76-87, June 2003

[41]
http://maude.cs.uiuc.edu/maude2-manual/html/index.html, Maude Manual (Version 2.3), 13/07/2007

[43]
http://www.microsoft.com/china/msdn/vstudio/, Microsoft Visual Studio Developing centre, 17/05/2007

[44]
http://msdn2.microsoft.com/en-us/library/aztlzleh.aspx, MSDN, XML documentation Sample, 17/05/2007

[45]
Design by Contract, B. Meyer, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986

[46]
Design by Contract, in Advances in Object-Oriented Software Engineering, D. Mandrioli and B. Meyer, Prentice Hall, 1991

[47]
Applying "Design by Contract", B. Meyer, in Computer (IEEE), 25, 10, October 1992

[48]
Object-Oriented Software Construction, B. Meyer, second edition. Prentice Hall, 1997

[49]
Algebraic foundations of systems specification, Donald Sannella and Andrzej Tarlecki, IFIP state-of-the-art reports, Springer, 1999

[50]
Initial algebra semantics and continuous algebras, J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. JACM, January 1977.

[51]
Lecture notes on the algebraic specification of data types, E. G. Wagner, Research Report RC 9203 39787, Mathematical Sciences Center, IBM Thomas J. Watson Research Centre, Yorktown Heights, New York, 1981.

[52]
Algebraic Methods in Semantics, J. Meseguer and J. A. Goguen, pages 459–541. Cambridge University Press, 1985.

[53]
Fundamentals of algebraic specification i: Equations and initial semantics, H. Erhig and B. Mahr, In EATCS Monograph, volume 6. Springer-Verlag, 1985.

[54]
Handbook of Logic in Computer Science, K. Meinke and J. V. Tucker., pages 189–411. Oxford University Press, 1992.

[55]
Universal algebra for computer scientists, W. Wechler, In EATCS Monograph, Springer-Verlag, 1991.

[57]
Classical Logic I: First Order Logic, the Blackwell Guide to Philosophical Logic,

Blackwell, 2001

[58]
http://www.kuro5hin.org/story/2005/7/29/04553/9714, *A case for formal specification*, Coryoth, 2005

[59]
A Specification Language, in On the Construction of Programs, Jean-Raymond Abrial, Stephen A. Schuman and Bertrand Meyer, Cambridge University Press, 1980

[60]
The Z notation: a reference manual, M. Spivey, Prentice Hall, 2001

[61]
Documentation of Software Products, J. D. Lomax, National Computing Centre publications, 1977

[62]
Literate Programming, Donal E. Knuth, Stanford, Califonia, Center for the study of language and information, 1992

[63]
CASL User Manual, Introduction to Using the Common Algebraic Specification Language, Michel Bidoit and Peter D. Mosses, Springer, 2004

[64]
A literate programming design language, Marcus E. Brown and David Cordes, In COMPEURO'90: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering, May 8-10, 1990, Tel-Aviv, Israel, pages 548-549. IEEE CS Press, Los Alamitos,CA, USA, 1990.

[65]
Programming pearls -- literate programming, Jon Bentley, Communications of the Association for Computing Machinery 364-369, CODEN CACMA2, May 1986

[66]
CASL Reference Manual, Reter D. Mosses, Springer, 2004

[67]
ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Uers, Guildford, UK, April 2005, Helen Treharne, Steve King, Martin Heson and Steve Schneider (Eds.), Springer, 2005

[68]
Formal Specification and Documenting Using Z, A Case Study Approach, Jonathan Bowen, International Thomson Publishing, 1996

[69]
http://www.w3.org/MarkUp/SGML/, W3C web site, Overview of SGML, 17/05/2007

[70]
The B-Book: Assigning Programs to Meanings, Jean-Raymond Abrial, Cambridge University Press, 1996

[71]
Specification in B: An Introduction using the B Toolkit, Kevin Lano, Imperial College Press, 1996

[72]
The Theory and Practice of Concurrency, A. W. Roscoe, Prentice Hall, 1998

[73]
Communicating Sequential Processes, C. A. R. Hoare, Prentice Hall, 1985

[74]
A Calculus of Communicating Systems, Robin Milner, Springer. 1982

[75]
Communicating and Mobile Systems: the Pi-Calculus, Robin Milner, Springer, 1999

[76]
http://msdn2.microsoft.com/en-us/library/system.collections.arraylist(vs.71).aspx , ArrayList Class library, 17/05/2007

[77]
http://java.sun.com/j2se/javadoc/, Javadoc Tool Homepage, 17/05/2007

[78]
Simula Begin, G.M. Birtwhistle, O.J.Dahl, B. Myhrhaug and K.Nygaard, Publisher Chartwell-Bratt Ltd. 1979

[79]
Météor: A Successful Application of B in a Large Project, Patrick Behm, Paul Benoit, Alain Faivre and Jean-Marc Meynadier, Publisher Springer Berlin / Heidelberg, 1999

[80]
http://msdn2.microsoft.com/en-us/library/w86s7x04(VS.80).aspx, MSDN, Using Properties (C# Programming Guide) , 17/05/2007

[81]
JML Reference Manual, Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joseph Kiniry., February 2007

[82]
Specifications Are (Preferably) Executable, N E Fuchs, Software Engineering Journal, Vol 7, No 5, pp323-334, Sept 1992

[83]
Specifications are not (necessarily) executable, I Hayes and C Jones, Software Engineering Journal Vol 4, No 6, pp330-338, Nov 1989

[84]
Types and Programming Languages, Benjamin C. Pierce, the MIT Press, 2002, ISBN 0-262-16209-1

[85]
An axiomatic basis for computer programming, C. A. R. Hoare, Communications of the ACM, 12(10):576–585, October 1969

[86]
Guarded commands, nondeterminacy and formal derivation of programs, Edsger W. Dijkstra, Communications of the ACM, 18(8):453–457, August 1975

[87]
Refinement Calculus: A Systematic Introduction, Ralph-Johan Back and Joakim von Wright, Springer, 1998