Algebraically modelling object-orientated programs.

Biddle, Justin

Algebraically Modelling Object-Oriented Programs

Justin Biddle BSc. (Wales)

A thesis submitted to the University of Wales in candidature for the degree of Philosophiae Doctor

Department of Computer Science
University of Wales, Swansea

June 2006

Summary

We explore the process of building algebraic models of the behaviour of *Java classes.* A fundamental building block of object-oriented programs is the *class* that can typically contain multiple *fields*, *constructors*, and *methods.* In Java a programmer can control access to the various methods, fields and constructors of a class. We will be *formally specifying* and *documenting* a class' public behaviour *algebraically.*

In practice even a simple Java class can have complex behaviour. A *full algebraic specification* (FAS) of a class can be complicated and hard to understand for someone who wishes to quickly ascertain the behaviour of a class. This complexity is largely as a result of machinery needed to define class behaviours that are implicit, that is behaviour that is considered part of Java's general language behaviour and defines the general structure of classes, methods, fields and constructors. However, it is unreasonable to expect a programmer to write such full specifications. Therefore we introduce the concept of an *Algebraic Class Specification* (ACS) that provides a much reduced version of the FAS of a class. The ACS is therefore more readable and is aimed at showing what we consider to be key information in the specification of a class that cannot be programmatically inferred from the language definition. Using the ACS we present a methodology for generating an FAS thus reducing the complexity of specification for the user.

We will show that the ACS provides a reader with a clear formal understanding of a class' behaviour using a minimum of information. The ACS is designed to be human readable yet still *machine readable.* We will show that in order to aid users in creating specifications of classes we have mimicked the Java syntax closely in the specification syntax. We will, in addition, present a methodology for embedding the formal semantic description for a Java class within *javadoc* comments thus allowing Java API documentation to contain both a formal specification of the behaviour of a class and its components and an informal general textual description.

These techniques have been developed by the analysis of case studies. We will demonstrate all of these techniques applied to a wide and varied range of both invented and existing examples of Java classes.

Declaration

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

Signed .... (candidate)

Date 03/07/2006

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)

Date 03/07/2006

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.

Signed .... (candidate)

Date 03/07/2006

Acknowledgements

This thesis would not have been possible without the help and encouragement of a number of people. Firstly I must thank my supervisor Dr. Neal Harman. Not only his knowledge has been invaluable but also without his guidance on direction, style, and motivation I would not be where I am today. He has been patient with me even when I know I have been at times too head strong and always helped me to see the correct course of action to take. I would also like to thank Professor John Tucker who has also been invaluable with his insight into my work and providing his tremendous experience to the structure and style of this thesis.

I would like to thank the University for their 3 year grant and putting their trust in me to complete this work. Without their financial aid this doctorate would not have been possible. I would also like to thank my employers CGram Software, not only for providing me with a wage to support me through my final year but also for their incredible accommodation and patience in allowing me the time to finish my thesis and to meet with my supervisor on a regular basis.

David Clark has been a great friend whose sense of humour and honest opinion have kept me going and helped me to see sense whenever I began to loose focus. Without his friendship it would have been a long and difficult fours years. Gareth Daniel like David is another of my closest friends who has always been there for me. His support and encouragement have helped make sure I never gave up no matter how difficult things became. Without Gareth's help I would not have become the person I am today and for that I am eternally grateful. Finally of my three closest friends there is Leah Clark. Leah has always been there to cheer me up and to support me whenever I needed it. Leah has been an example to me of someone who perseveres with something no matter how difficult things become and I have tried to follow her example. A thank you must also go to all the other post-graduates who have been there during my studies. A day in the lab was never dull with you around. I also wish to thank Chris Whyley who despite my best efforts retained his sanity and good humour despite almost relentless attempts to rob him of them.

I wish to thank all the members of my family for their support especially my Mum, Dad, Anth, and Marianne whose moral and financial support have been invaluable. Thank you for always being there for me. I also wish to thank Gerry, Diane and Naomi Shaw who helped provide a family environment down here in Swansea for the final year of my PhD. It has been a joy to spend time with you all especially during this difficult final year.

Finally I could not have completed this thesis without the faith and strength of my Saviour, Jesus Christ. Without my faith I would not have had the strength to have completed this thesis and as with all things I do, they are to honour him.

Contents

Chapter 1

Introduction

This thesis explores the process of building algebraic models of the behaviour of *Java classes.* A fundamental building block of object-oriented programs is the *class* which for now can be approximated as a datatype that can typically contain multiple *fields*, *constructors*, and *methods.* In Java a programmer can control access to the various methods, fields and constructors of a class. In this thesis we are specifically interested in those parts of the class that are *public.* That is the fields, methods, and constructors that programmers can access externally with their own classes and programs. In particular, we are interested in *formally specifying* and *documenting* a class' public behaviour *algebraically* [MT92].

We have chosen to specify classes algebraically because of the long history of research by workers currently in the Algebraic Methods Group at the University of Wales Swansea, such as [BT87, BT93, Ree01, STR03, Ste96, Har00, Har02]. In practice even a simple Java class can have complex behaviour. For example, consider the following class.

```
public class AClass{

    public AClass(){
    }

    public int return4(){
        return 4;
    }
}
```

One possible algebraic model of this class (written in *Maude* [SRI05]) is shown below.

```
fmod AClass is

    protecting BUILDLINK .

    sort AClass .
    sort AClassInt .

    op AClass() :  -> AClass .

    op AAClass : -> AClass .

    op _,_ : AClass Int -> AClassInt .
    op oval(_) : AClassInt -> AClass .
    op qval(_) : AClassInt -> Int .
    op _.return4() : AClass  -> AClassInt .
    op _.return4() : AClassInt  -> AClassInt .
    op _.return4()o : AClass -> AClass .
    op _.return4()q : AClass -> Int .


    var A : AClass .
    var aclass : AClass .
    var int : Int .
    var aclassint : AClassInt .

    eq AClass() = AAClass .
    eq (A).return4()q = 4 .
    eq oval(aclass,int) = aclass .
    eq qval(aclass,int) = int .
    eq (aclass).return4() = (aclass).return4()o,(aclass).return4()q .
    eq (aclassint).return4() =
        (oval(aclassint)).return4()o,(oval(aclassint)).return4()q .

endfm
```

The above example is a simplified version of an algebraic specification modelling the above Java class. For example, among other simplifications, every Java class inherits from another class called `Object`, however the example above omits this.

Some important points about the algebraic specification of `AClass` above are as follows:

1. `protecting BUILDING` imports a series of Maude modules containing predefined functionality such as the sort `INT` (see appendix A).

2. `sort AClassInt` This defines a tuple type which is returned by methods to allow them to return both a value and the updated instance of the class.

3. The tupling operator `_,_` which combines an *AClass* instance and an integer value to produce a tuple of the type `AClassInt`.

4. Operators `oval` and `qval` and equations. These are used to project the value part `qval` and the new class instance part `oval` of the tuple type.

5. There are two definitions of `return4`. One takes in the class type, `AClass`, as an input, the other takes in the tuple type, `AClassInt`, and both return the tuple type, `AClassInt`. This is neccessary so as `return4` can be called on all possible tuple types as well as the class type.

6. We have two operators defining the behaviour of `return4`. The `return4()q` operator defines the query value and `return4()o` defines the new class instance returned by `return4`. However because method `return4` is a *query* (that is, it does not change the state of a class instance) we only define the behaviour of `return4()q` and omit the trivial equation that `return4()o` is the identity function.

7. Finally we have equations like the following:

   ```
   eq (aclass).return4() = (aclass).return4()o,(aclass).return4()q .
   ```

   These equations are used to build the tuple type *AClassInt* using `return4()q` and `return4()o` as the value returned by `return4`.

As can be seen the above example, the *full algebraic specification* (FAS) of a class can be complicated and hard to understand for someone who wishes to quickly ascertain the behaviour of a class even for the very simple example that we have chosen. This complexity is largely as a result of machinery needed to define class behaviours that are implicit, that is behaviour that is considered part of Java's general language behaviour and defines the general structure of classes, methods, fields and constructors. However, it is unreasonable to expect a programmer to write specifications like that of `AClass`. Therefore we will introduce the concept of an *Algebraic Class Specification*, ACS that provide a much reduced version of the FAS of a class. The ACS is therefore more readable and is aimed at showing what we consider to be key information in the specification of a class that cannot be programmatically inferred from the language definition. As will be seen in Section 5.3 we have created a tool called the *Algebraic Specification Generator* (ASG) that can generate an FAS from a class with embedded information representing an ACS. The following is an ACS for our example above.

```
Class AClass{

    Hidden{
        op AAClass : -> AClass .
    }

    Fields{
    }

    Constructors{
        AClass : .
    }

    Methods{
        op return4 : -> Int .
    }

    Operations{
    }

    Variables{
        var A : AClass .
    }

    Equations{
        eq AClass() = AAClass .
        eq (A).return4()q = 4 .
    }
}
```

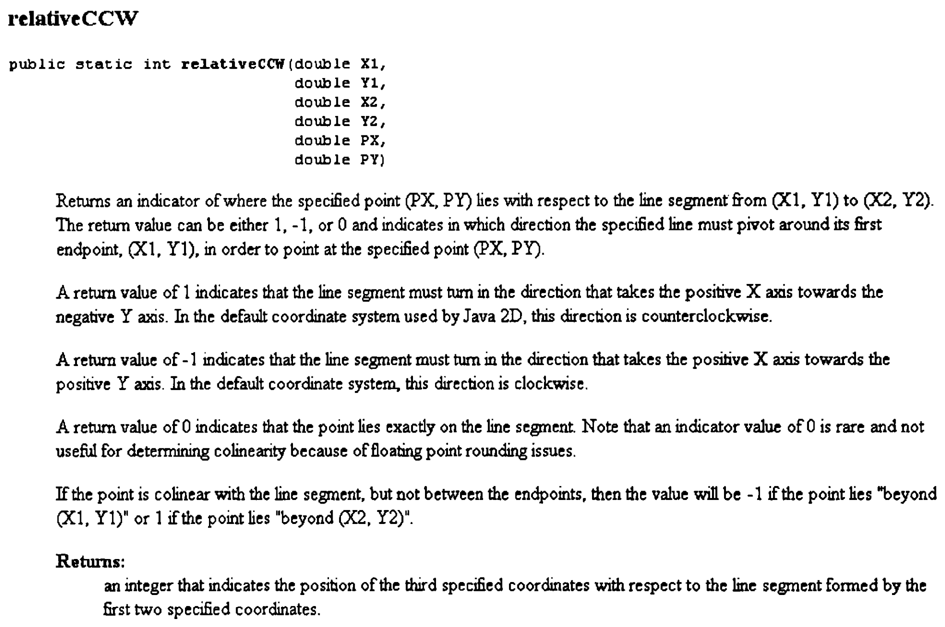
The ACS, we feel provides a reader with a clear formal understanding of a class' behaviour using a minimum of information. The ACS is designed to be more easily human readable yet still *machine readable.* A large part of this thesis (Chapter 4) is about how we generate an FAS from an ACS.

The specification techniques used throughout this thesis were developed experimentally by analysis of case studies which we examine in more detail in Chapter 6.

1.1 Specification and Documentation

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. All another programmer needs to know is the *public interface* of a class. They would then be able to utilise that class within their own programs. This is the basis of the notion of an *Application Program Interface* (API). However, although the syntax of the public interface of a class itself can and usually is documented in a reasonably formal way, the semantics is defined simply with natural language. Java in particular offers API documentation [Sun05g] 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 also allows other programmers to generate API documentation in the same format for their own classes by extracting documentation (and specification) information from special *javadoc* comments. However due to the informality of the semantic description this documentation can be often difficult to understand, ambiguous, or incorrect. We will now look at examples of each of these cases. The first two cases are actual examples of the Sun Java API. The third example is one of our own devising, used to show how an inexperienced user can incorrectly define the behaviour of a Java class using the informal documentation.

- Difficult to understand documentation. Consider the informal documentation for a method called `relativeCCW` from the class `Line2D` as shown in Figure 1.1. The informal textual description is quite long and in places difficult to follow. If we look at our formal documentation shown in Figure 1.2 the added equational definition given in the documentation clearly and precisely defines the behaviour of the method.



relativeCCW

```
public static int relativeCCW(double X1,
                              double Y1,
                              double X2,
                              double Y2,
                              double PX,
                              double PY)
```

Returns an indicator of where the specified point (PX, PY) lies with respect to the line segment from (X1, Y1) to (X2, Y2). The return value can be either 1, -1, or 0 and indicates in which direction the specified line must pivot around its first endpoint, (X1, Y1), in order to point at the specified point (PX, PY).

A return value of 1 indicates that the line segment must turn in the direction that takes the positive X axis towards the negative Y axis. In the default coordinate system used by Java 2D, this direction is counterclockwise.

A return value of -1 indicates that the line segment must turn in the direction that takes the positive X axis towards the positive Y axis. In the default coordinate system, this direction is clockwise.

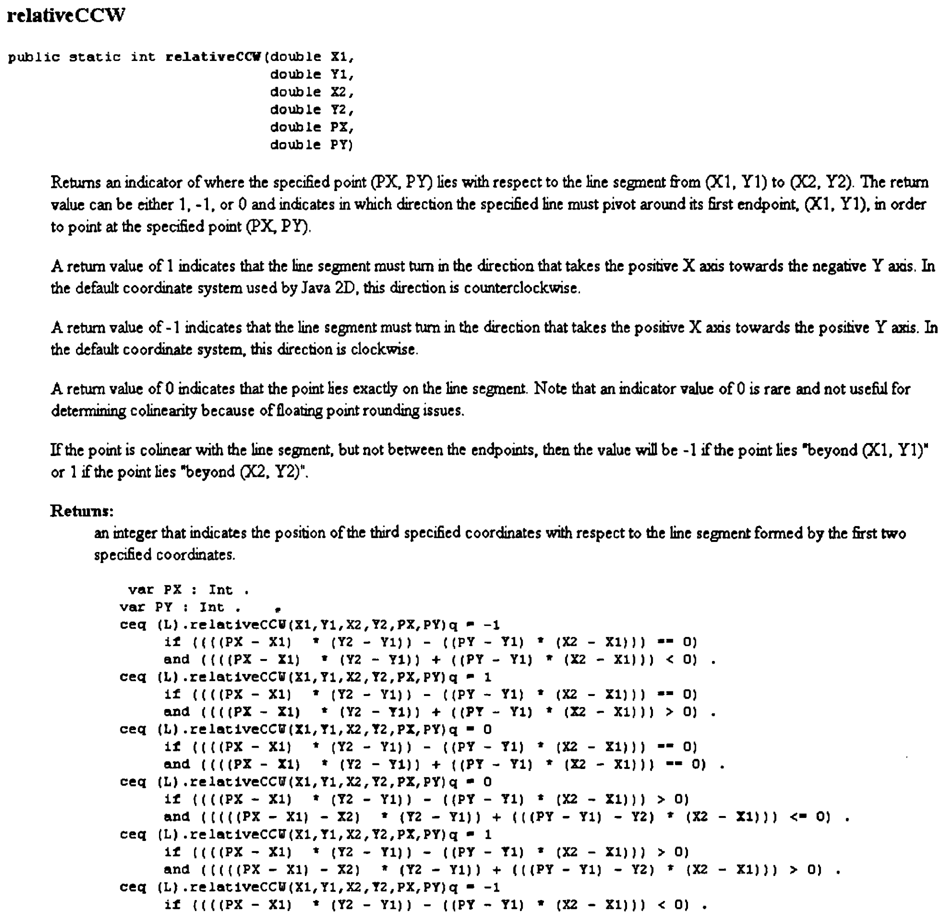
A return value of 0 indicates that the point lies exactly on the line segment. Note that an indicator value of 0 is rare and not useful for determining colinearity because of floating point rounding issues.

If the point is colinear with the line segment, but not between the endpoints, then the value will be -1 if the point lies "beyond (X1, Y1)" or 1 if the point lies "beyond (X2, Y2)".

Returns:
an integer that indicates the position of the third specified coordinates with respect to the line segment formed by the first two specified coordinates.

Figure 1.1: `relativeCCW` Informal API Documentation.

- Ambiguous Documentation. Consider the informal documentation for the methods called `distance` and `distanceSq` from the class `Point2D` as shown in Figure 1.3. Due to the informality of the documentation, the actual behaviour of the methods is ambiguous. Informally it might be read that `distance` performs the full calculation for the distance and that `distanceSq` is calculated by squaring the value returned by `distance`. However this is not in fact the case. Due to the fact that `distance` can also be viewed as being the square root of `distanceSq` to calculate the two methods in the manner discussed above will result in rounding errors when finding a square root and then squaring it again. If the original code is examined then it is seen that `distance` is actually calculated by taking the square root of `distanceSq`. Therefore the informal documentation is ambiguous about the true behaviour of the methods which could lead to inaccurate results being generated. However if we look at the formal documentation in Figure 1.4 the behaviour of both methods is clearly defined by the formal equations and removes the ambiguity of the informal documentation (note the `sqr` and `isqrt` are built in functions used to calculate the square and the square root of a given number respectively).



relativeCCW

```
public static int relativeCCW(double X1,
                              double Y1,
                              double X2,
                              double Y2,
                              double PX,
                              double PY)
```

Returns an indicator of where the specified point (PX, PY) lies with respect to the line segment from (X1, Y1) to (X2, Y2). The return value can be either 1, -1, or 0 and indicates in which direction the specified line must pivot around its first endpoint, (X1, Y1), in order to point at the specified point (PX, PY).

A return value of 1 indicates that the line segment must turn in the direction that takes the positive X axis towards the negative Y axis. In the default coordinate system used by Java 2D, this direction is counterclockwise.

A return value of -1 indicates that the line segment must turn in the direction that takes the positive X axis towards the positive Y axis. In the default coordinate system, this direction is clockwise.

A return value of 0 indicates that the point lies exactly on the line segment. Note that an indicator value of 0 is rare and not useful for determining colinearity because of floating point rounding issues.

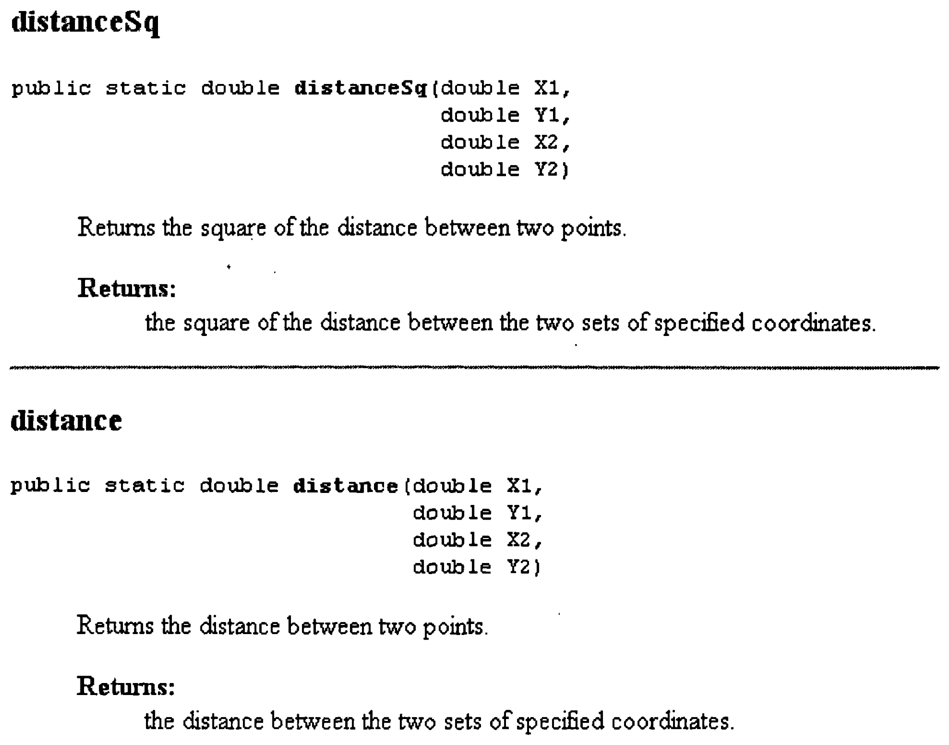
If the point is colinear with the line segment, but not between the endpoints, then the value will be -1 if the point lies "beyond (X1, Y1)" or 1 if the point lies "beyond (X2, Y2)".

Returns:

an integer that indicates the position of the third specified coordinates with respect to the line segment formed by the first two specified coordinates.

```
 var PX : Int .
var PY : Int .
ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = -1
     if ((((PX - X1)  * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1))) == 0)
     and ((((PX - X1)  * (Y2 - Y1)) + ((PY - Y1) * (X2 - X1))) < 0) .
ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 1
     if ((((PX - X1)  * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1))) == 0)
     and ((((PX - X1)  * (Y2 - Y1)) + ((PY - Y1) * (X2 - X1))) > 0) .
ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 0
     if ((((PX - X1)  * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1))) == 0)
     and ((((PX - X1)  * (Y2 - Y1)) + ((PY - Y1) * (X2 - X1))) == 0) .
ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 0
     if ((((PX - X1)  * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1))) > 0)
     and (((((PX - X1) - X2)  * (Y2 - Y1)) + (((PY - Y1) - Y2) * (X2 - X1))) <= 0) .
ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 1
     if ((((PX - X1)  * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1))) > 0)
     and (((((PX - X1) - X2)  * (Y2 - Y1)) + (((PY - Y1) - Y2) * (X2 - X1))) > 0) .
ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = -1
     if ((((PX - X1)  * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1))) < 0) .
```

Figure 1.2: relativeCCW Formal API Documentation.



distanceSq

```
public static double distanceSq(double X1,
                                double Y1,
                                double X2,
                                double Y2)
```

Returns the square of the distance between two points.

Returns:
the square of the distance between the two sets of specified coordinates.

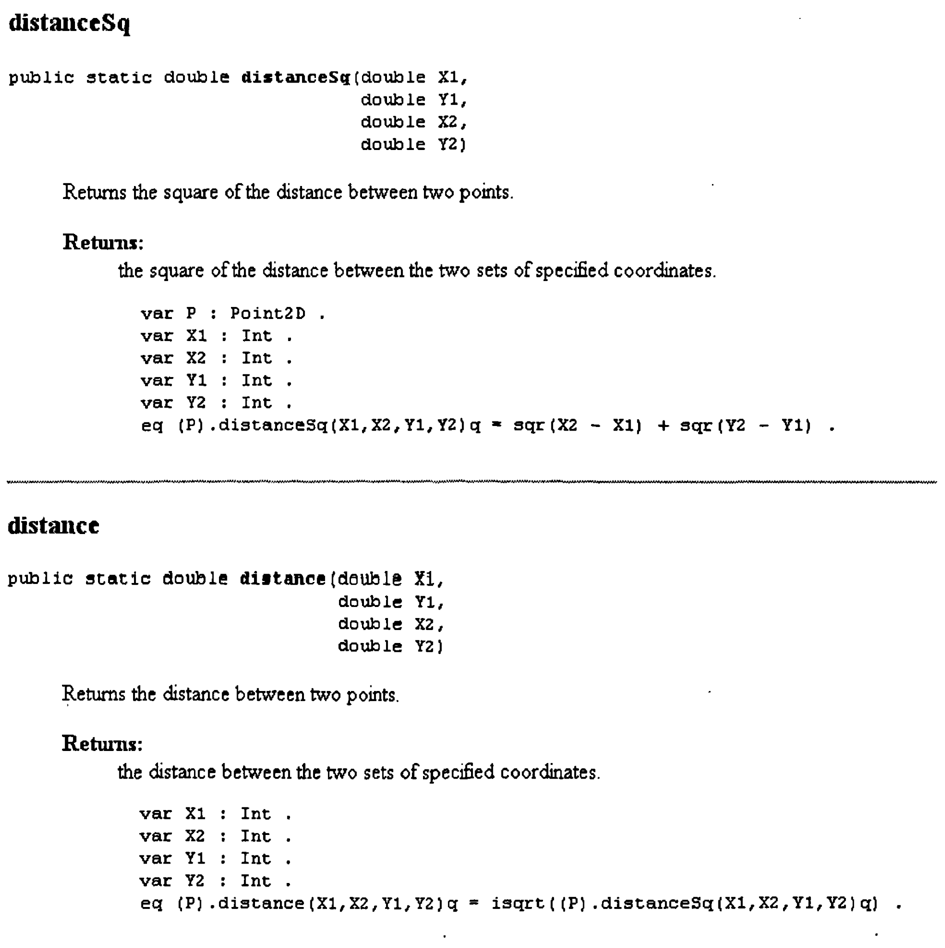
distance

```
public static double distance(double X1,
                              double Y1,
                              double X2,
                              double Y2)
```

Returns the distance between two points.

Returns:
the distance between the two sets of specified coordinates.

Figure 1.3: `distance` and `distanceSq` Informal API Documentation.



distanceSq

```
public static double distanceSq(double X1,
                                double Y1,
                                double X2,
                                double Y2)
```

Returns the square of the distance between two points.

Returns:
the square of the distance between the two sets of specified coordinates.

```
var P : Point2D .
var X1 : Int .
var X2 : Int .
var Y1 : Int .
var Y2 : Int .
eq (P).distanceSq(X1,X2,Y1,Y2)q = sqr(X2 - X1) + sqr(Y2 - Y1) .
```

---

distance

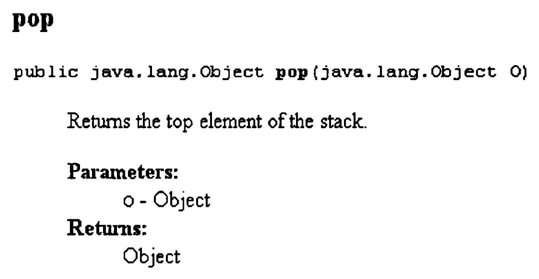
```
public static double distance(double X1,
                              double Y1,
                              double X2,
                              double Y2)
```

Returns the distance between two points.

Returns:
the distance between the two sets of specified coordinates.

```
var X1 : Int .
var X2 : Int .
var Y1 : Int .
var Y2 : Int .
eq (P).distance(X1,X2,Y1,Y2)q = isqrt((P).distanceSq(X1,X2,Y1,Y2)q) .
```

Figure 1.4: `distance` and `distanceSq` Formal API Documentation.



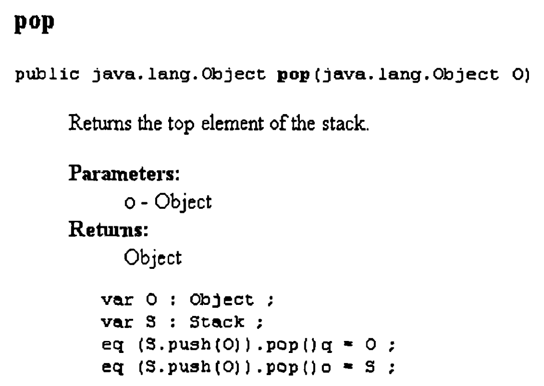
pop

```
public java.lang.Object pop(java.lang.Object O)
```

Returns the top element of the stack.

Parameters:
o - Object
Returns:
Object

Figure 1.5: pop Informal API Documentation.



pop

```
public java.lang.Object pop(java.lang.Object O)
```

Returns the top element of the stack.

Parameters:
o - Object
Returns:
Object

```
var O : Object ;
var S : Stack ;
eq (S.push(O)).pop()q = O ;
eq (S.push(O)).pop()o = S ;
```

Figure 1.6: pop Formal API Documentation.

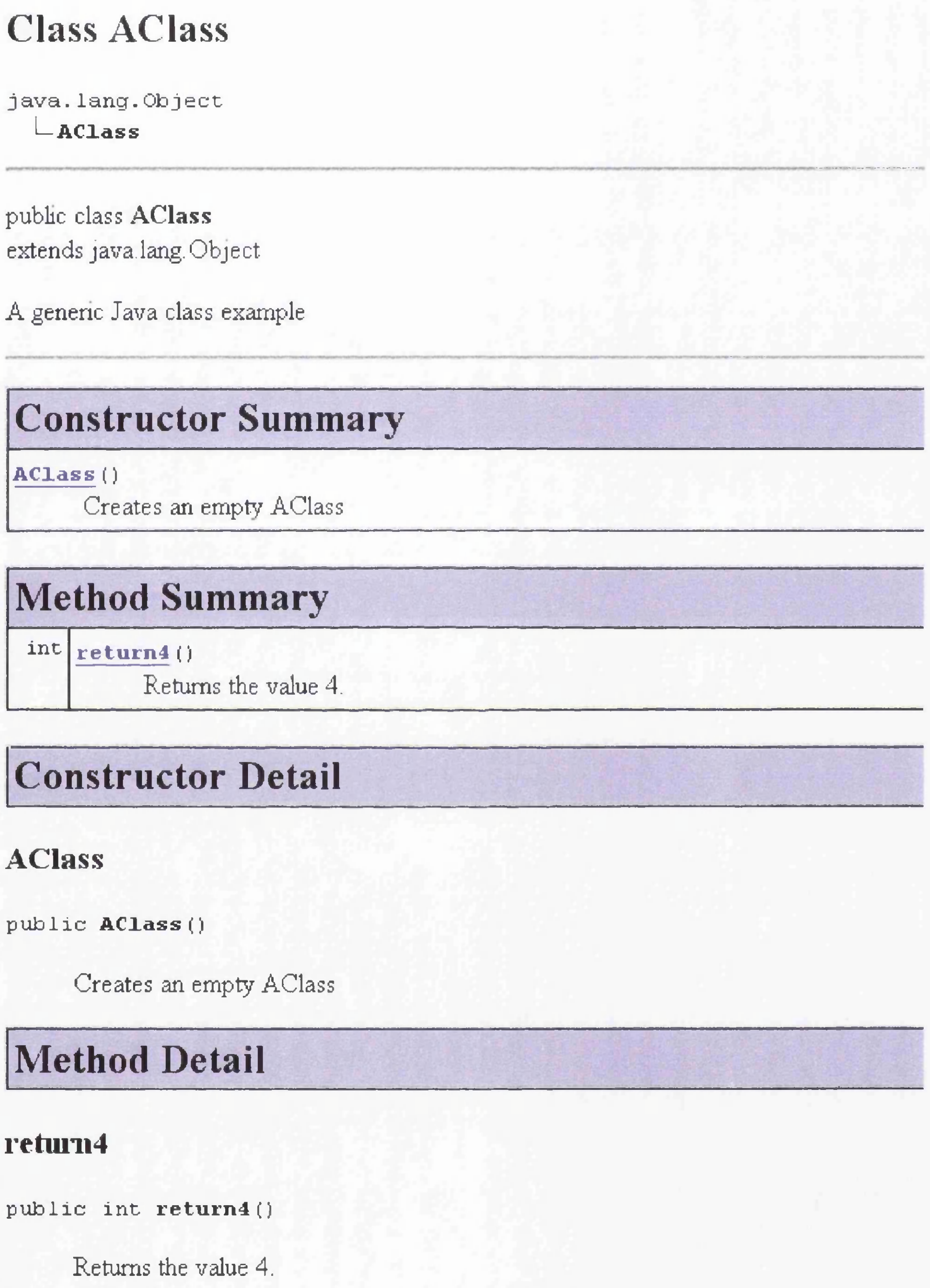
- Incorrect documentation. Consider the informal documentation for the method called pop (as shown in Figure 1.5) from a class `Stack` which can be assumed to define the obvious behaviour for a stack. The behaviour of the method is defined as returning the top value of the stack. However, in this case pop also removes the top element from the stack at the same time. Although this behaviour is usually assumed to be the standard behaviour for popping an element of a stack, the informal documentation does not explicitly state this and therefore it is incorrect. A user might read the informal documentation and decide to call the pop method to obtain the top element of the stack and not realise that the state of the stack has been changed as well. If we look at the formal definition of the pop method as shown in Figure 1.6 the behaviour is fully defined by the formal equations and therefore the documentation is now correct.

As seen above in order to try and solve this problem we will show in this thesis how we can embed the formal semantic description for a Java class within the *javadoc* comments thus allowing the API documentation to contain both a formal specification of the behaviour of a class and its components and an informal general textual description (see Section 5.3). The purpose of the documentation is as follows:

- To allow a developer who is uncertain of the behaviour of the informal documentation to be able to refer to the concrete formal specification. This formal specification will provide a clear and precise definition of the class it is defining. For example, the informal specification for our `AClass` example using standard Javadoc comments is shown in Figure 1.7. Using our formal documentation which is embedded within the informal documentation we get a clear and precise definition of the behaviour of `AClass` as shown in Figure 1.8 (note we have only shown those bits of the new documentation which contain the formal documentation and omit the parts that remain the same as the standard informal documentation to avoid repetition).
- The formal specification can be extracted into an executable Maude specification. A user will also be able to execute the specification to test how a class is meant to behave and to use their own specifications to experiment with how a class will interact with their own user defined classes.
- A future use of our documentation would be to allow users to extract the formal specification into the executable Maude format to allow them to test it against the actual Java class it specifies. This can be used to verify if the implementation of Java class behaves as specified.

1.2 Automatically Building ACS and FAS specifications

In order to aid the user we have implemented a program called the *Algebraic Specification Generator* (ASG) that can take in a Java class with extra embedded equations and can use this to generate both the ACS and FAS specifications. The tool was primarily built to aid in the testing of new concepts and refinements to our model as discussed in section 5.3. However the tool can also be used to aid in the automatic generation of the ACS and FAS specification. As the code is long we will not be examining the full program



Class AClass

```
java.lang.Object
  └AClass
```

public class **AClass**
extends java.lang.Object

A generic Java class example

Constructor Summary
`AClass()` Creates an empty AClass

Method Summary	
`int`	`return4()` Returns the value 4.

Constructor Detail

AClass

```
public AClass()
```

Creates an empty AClass

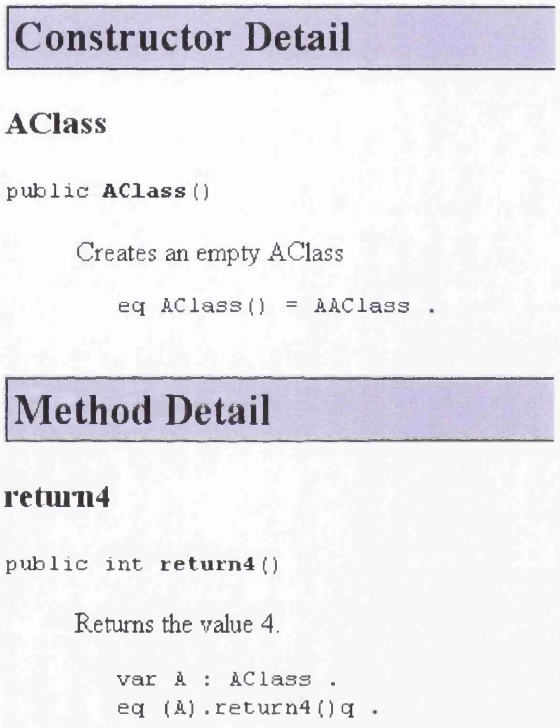
Method Detail

return4

```
public int return4()
```

Returns the value 4.

Figure 1.7: `AClass` Informal API Documentation.



Constructor Detail

AClass

```
public AClass()
```

Creates an empty AClass

```
eq AClass() = AAClass .
```

Method Detail

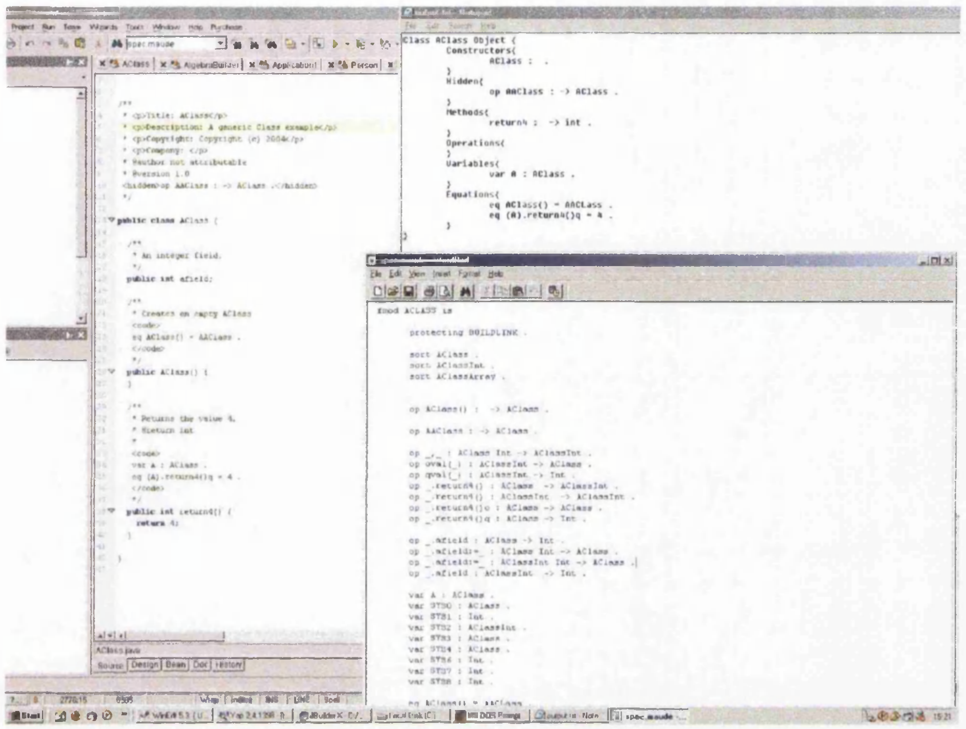
return4

```
public int return4()
```

Returns the value 4.

```
var A : AClass .
eq (A).return4()q .
```

Figure 1.8: `AClass` New Formal API Documentation.



```
/**
 * <p>Title: AClass</p>
 * <p>Description: A generic Class example</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 <hidden>op AAClass : -> AClass .</hidden>
 */

public class AClass {

  /**
   * An integer field.
   */
  public int afield;

  /**
   * Creates an empty AClass
   <code>
   eq AClass() = AAClass .
   </code>
   */
  public AClass() {
  }

  /**
   * Returns the value 4.
   * @return int
   *
   <code>
   var A : AClass .
   eq (A).return4()q = 4 .
   </code>
   */
  public int return4() {
    return 4;
  }

}
```

```
Class AClass Object {
        Constructors(
                AClass :  .
        }
        Hidden(
                op AAClass : -> AClass .
        }
        Methods(
                return4 :  -> int .
        }
        Operations(
        }
        Variables(
                var A : AClass .
        }
        Equations(
                eq AClass() = AAClass .
                eq (A).return4()q = 4 .
        }
}
```

```
fmod ACLASS is

        protecting BUILDLINK .

        sort AClass .
        sort AClassInt .
        sort AClassArray .


        op AClass() :  -> AClass .

        op AAClass : -> AClass .

        op _,_ : AClass Int -> AClassInt .
        op oval(_) : AClassInt -> AClass .
        op qval(_) : AClassInt -> Int .
        op _.return4() : AClass  -> AClassInt .
        op _.return4() : AClassInt  -> AClassInt .
        op _.return4()o : AClass -> AClass .
        op _.return4()q : AClass -> Int .

        op _.afield : AClass -> Int .
        op _.afield:=_ : AClass Int -> AClass .
        op _.afield:=_ : AClassInt Int -> AClass .
        op _.afield : AClassInt  -> Int .

        var A : AClass .
        var STS0 : AClass .
        var STS1 : Int .
        var STS2 : AClassInt .
        var STS3 : AClass .
        var STS4 : AClass .
        var STS6 : Int .
        var STS7 : Int .
        var STS8 : Int .

        eq AClass() = AAClass .
```

Figure 1.9: Examples of an input class (AClass) listing to the ASG and its outputs for the ACS and FAS.

listings. Instead we will discuss its basic structure and examine example sections of the codes in later chapters. The full code can be found on the appendix CD and instructions on its use can be found in appendix A. An example of input and outputs for the ASG can be seen in Figure 1.9.

1.3 Executable Specifications

As already stated, our algebraic specification is written in *Maude* [SRI05] allowing us to make it executable. We do this because we feel that it would be beneficial for the specification to be run and to be able to test the behaviour of the formal specification. However it is not a requirement of our algebraic specification that they should be written in Maude and is simply the language which we have chosen. Other specification languages such as CASL [CAF05c] could be used instead. A consequence of our choice of Maude is that we use the *initial model* [MT92] for algebraic specification.

1.4 Ease Of Use

One of our aims is that our formal specifications should be easy to use and understand for a Java programmer, if they are to be usable in practice. It is important therefore that the specification notation follows the Java syntax as much as possible. It is also important that the equations that are a consequence of the Java language definition (not the equations used to define the semantics of the methods and constructors themselves) should be automatically generated. As will be seen later we have designed a tool which we used in order to design and test new concepts for our specification model which could be adapted to be used for the purpose of automatically generating FASs for user defined classes.

1.5 Specifying Functionality

In our work we try and specify a broad range of Java's functionality. It is our aim to provide a strong basis for specifying Java that in future work can be extended and improved upon. Java is a large language with a lot of built-in classes. It would therefore to be impossible to model the whole language in complete detail in the scope of this thesis. Rather it is our aim to model a strong general model that specifies what we consider to be the core functionality of Java. We will also attempt to model functionality that we consider interesting such as Java's reflection classes. Reflection is used by our automated tool and therefore we feel it is important for a concise model to show how these classes are modelled algebraically.

1.6 Overview of Thesis

The structure of this thesis is as follows:

In Chapter 2 we will examine the background to our research. We will look at the history of object-oriented languages with particular attention paid to C#, C++ and Java. We will then examine algebraic specification and different types of algebras such as *many-sorted* and *order-sorted* algebra. We will look at how under certain conditions order-sorted algebra can be considered to be equivalent to many-sorted algebras. We will also look at some practical application of algebraic specification and the different tools available. Finally we will look at some of the work done in formally specifying object-oriented concepts, programs, and languages.

In Chapter 3 we will look at the process of algebraically modelling the publicly visible parts of Java classes. We will examine the key features of a Java class (*methods*, *fields*, *constructors*, and *inheritance*). We will then introduce the concepts of *algebraic interfaces* and *algebraic class specification* (ACS) for Java and use these concepts to create a *full algebraic specification* (FAS) of a Java class. We will then look at how our works relates to and builds on the work of [STR03] on interfaces and libraries. Finally we will look at the code we have written for automatically building an ACS specification from a Java class.

In Chapter 4 we will show how we will show how to extend an ACS into an FAS which will require many extra equations and operators to fully define the class. 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 looking at modelling classes with several levels of inheritance. These FASs will be written in the *Maude* language to allow us to produce an executable specification. We will then again examine how this work relates to and builds on the work of [STR03]. Finally we look at the parts of the code we have written for automatically building an FAS specification from a Java class.

In Chapter 5 we will extend the functionality of our model with extra Java functionality that we have pre-defined. We will look at modelling arrays and how to generate equations for arrays. In addition we will model *reflection* and will examine it in some detail. We do this as we feel that reflection is an important part of a concise model of Java object-oriented programs. We will also look at the automated conversion tool we have written which was used to help develop our model. The tool can take a Java class with appropriate equations embedded in the Javadoc comments and convert it into an executable FAS written in Maude. The user is only required to define the semantic equations for the behaviour of each method and constructor.

In Chapter 6 we will look at examples of specifying Java classes from both the Java API and from user defined classes. We will look at interesting aspects that we encountered when modelling them and how we solved particularly interesting problems in modelling some of the semantics. Finally we will examine particular problems that our model as yet cannot specify and suggest ways in the future that they could be solved and the model improved.

Finally in Chapter 7 we will summarise our achievements and suggest areas where further work can be done (besides those concerning Java functionality mentioned in Chapter 6) to build upon the work we have presented here.

Chapter 2

An Overview of Object-Oriented Programming and Algebraic Specification

This chapter surveys existing work in the field of *formal specification* and, in particular, *algebraic specification* and its application to *Object-Oriented Programming.* We will briefly examine the history of *object-oriented* programming and in particular Java, which is the object-oriented programming language that we have chosen as the basis of our model. We will also examine the background of algebraic specification and its various forms. We will look at existing work on Java and other object-oriented programming languages in the field of formal specification.

The structure of this chapter is as follows. Section 2.1 will look at the history of object-oriented programming and some of the main languages available. We will discuss the history of these main languages and also examine their contributions to object-oriented programming. We will look in most detail at the language of Java as this is the object-oriented language that we have chosen to model. Section 2.2 will examine the background of algebraic specification and some of the important forms of algebraic specification available. We will look at the different types of algebra that we use in our model, chiefly *many-sorted* and *order-sorted* algebras. We will also show how, under certain conditions, order-sorted algebras can be shown to be equivalent to many-sorted algebras which is important for our work. We will look at several specification languages but focus in more detail on three important algebraic languages CASL, Maude (and its predecessor OBJ), and Larch. We will look at Maude in the most detail as this is the specification language we have chosen to use. We will also look at a variety of ways in which algebraic specification has been used (although uses relating specifically to object-oriented programming will be discussed in Section 2.3). Finally in Section 2.3 we will examine some of the major areas in which formal specifications have been successfully applied to that of object-oriented programming. In particular we are interested in formally specifying object-oriented languages and Java using algebraic methods. We will examine three key pieces of research that relate to our own, the most relevant being a way of automatically generating algebraic equations for Java classes. We will briefly discuss other interesting works in the field of formally specifying object-oriented programs.

2.1 Object-Oriented Programming

In this section we will give a brief overview of the history of object-oriented programming and look at the current state of object-oriented programming languages. We will pay particular attention to Java as this is the language that we wish to specify. Because object-oriented programming is a broad topic, we will restrict ourselves to areas we consider to be the important background for this thesis. Our work is primarily interested in the modelling of the public interfaces of classes. We will be focusing on the modelling of the concepts of a *class*, a *class instance*, a *constructor*, a *method*, a *field*, and the behaviour of *class inheritance*.

As object-oriented programming is broad and varied we will choose in the thesis to model only those concepts which we consider to be core to the object-oriented paradigm. We are principally interested in *Classes*. The core features of object-oriented classes are a collection of *methods*, *fields*, and *constructors*. Fields are a variables that specifically belong to the class. They can only be accessed through an instance of the class (commonly called *objects*). Fields can either be primitive datatypes such as integers and floating point numbers or more complex datatypes such as arrays and class instances. There are different ways for handling fields. One way is enforce that all access to fields is done via functions that can be used to get and set the field's value. Another method is to allow full access to a field, allowing code external to the class to directly set and get the field's value. In Java a combination of these two approaches is used. Access to a field can be restricted or allowed by use of the *public*, *private*, and *protected* modifiers. As we are only interested in a classes public interface therefore we will only be modelling fields that can be accessed directly (those with the *public* identifier). Methods are functions which also specifically belong to the class. These can be used both to return a value and change the state of the class (by changing the class' field values). In Java, methods which do not return a value are called *void* methods and their return types are denoted as *void*. *Constructors* are special methods that are used to create a new instance of a class. At their basic level they simply create a new class with all fields set to those values which Java defines as the default initial value for a certain type. More complex constructors can be used to create new class instances and initialise a class' fields to specific values.

In addition to these core components of an object-oriented class we will also in this thesis model the behaviour of class *inheritance.* Using inheritance, classes can be defined as being more specialised versions of other classes. These inheriting classes inherit methods and fields are then considered as belonging to this new class as well. The inherited methods can be overridden by new method definitions if desired. In Java if a class overrides an inherited method, then the inherited methods original behaviour can be accessed via use of the *super* keyword (see Section 4.3 for a more detailed discussion of inheritance). Although some object-oriented languages permit a class to inherit from multiple classes, Java only allows single inheritance therefore we will only concentrate on single inheritance and will not be discussing the complications of multiple inheritance in this thesis.

In order to model complete object-oriented programs we would need to model the concept of an execution model [GM96, Har89, Ste96]. The basic principles of an execution model are as follows. The state of a system is altered by the iteration of a next state function.

$$\begin{aligned} F(0, a) &= init(a) \\ F(t+1, a) &= next(F(t, a)) \end{aligned}$$

Which is defined as follows.

- A is the set of all possible states of the system.
- T is the set of all possible steps of the system (often denoting units of time).
- $F : T \times A \longrightarrow A$ where F calculates the state of the system at a given step t. The element $t \in T$ is the current step in the system and $a \in A$ represents a state of the system.
- $init : A \longrightarrow A$ is used to initialise the state of the system.
- $next : A \longrightarrow A$ calculates the next state of the system based on the current state of the system.

An executable model of an object-oriented system would define the state of a program as being defined as the current state of all the objects in the system. Objects would be stored in a repository to which new objects can be added, objects can be changed, and objects can be removed from the repository. The state of each object would be defined in terms of its member variables (fields). A single step in the system is harder to define. The execution of a method belonging to one of the objects in the repository could result in the execution of many more methods, in many more objects before it is completed. In our model we are only interested in the local behaviour of a class. Our model will look at the behaviour of individual classes and single instances of them (occasionally with the addition of extra "helper" classes and instances when needed to fully test the behaviour of the individual class being modelled) and as such our model does not include an execution model. In Section 6.7.5 we will examine the complexities that would be involved if our model was adapted to model the concept of a program consisting of many individual classes and instances.

2.1.1 A Brief History of Object-Oriented Programming

The first example of object-oriented programming dates back to 1962 when development began on a language called SIMULA I which was completed in 1965. SIMULA I was an extension of the Algol 60 language [And64]. Although lacking many of the key features of an Object-Oriented language it introduced the concept of a *record class* which in turn would lead the way to Object-Oriented Programming Languages. This was followed in 1967 by SIMULA 67 [Bir72, DMN70]. This language is credited with introducing many of the main Object-Oriented programming principles we use today such as objects, classes and inheritance. In the 1970's Alan Kay's group at Xerox PARC developed Smalltalk [PW88] using SIMULA as a platform. This language added a graphical user interface and interactive program execution to the concepts of object-oriented programming. Although not widely used commercially anymore Smalltalk is still in limited use in some sectors of the industry. Between 1982 and 1983 Brad Cox and Tom Love developed Objective C [CN91, Koc03]. This language extends C [Ker88, Ber86] by adding classes, messages and inheritance. It is compatible with C and a C program can be compiled with an Objective C compiler. The extensions made to the C language are specifically based on Smalltalk with the object model in Objective C being similar to Smalltalk-76. The operating system *MAC OS X* [ACI05] is written in Objective C. Another important object-oriented language is Eiffel [Mey92b, Mey88, TW95] developed in 1985 by Bertrand Meyer. The language had many of the key concepts of object-oriented programming such as multiple inheritance (although we will see in section 2.1.2 the current view is that this is no longer considered to be good practice) and the idea of a class being a type. It also introduced the concept of *Design By Contract* [Mey92a, ESo05] which although not being an object-oriented programming concept was well suited to the programming style. The concept of 'Design By Contract' is based around *pre* and *post* conditions as defined in *Hoare Logic* [Hoa69, Ten02]. The main feature of this was the *Hoare Triple*

$$\{P\}\ C\ \{Q\}$$

The triple is used to describe how the computation state is changed upon the execution of a piece of code. C represents the command being executed and P and Q are assertions that have to hold before and after the execution of C. P is the *precondition* and Q is the *postcondition*. The above equation can be read as that whenever P holds before the execution of C then Q must hold after the execution, provided that C terminates. A simple example is as follows

$$\{x = 2\}\ y := x * 2\ \{x = 2\ \wedge\ y = 4\}$$

This is adapted to 'Design By Contract' by stating conditions that have to hold before and after the execution of a piece of code. The preconditions are user requirements and the user has to fulfil these before running the piece of code. An example might be that the user must not pass a *null* pointer to a method that requires a pointer to be passed. The postconditions are conditions placed on the routine that is being called and the routine guarantees that it will fulfil these conditions on completion. An example might be that the routine will always return a value greater than zero.

The first industrially successful object-oriented programming language was C++ [Str91, ES90]. Up to this point the difficulty of persuading the commercial sector to use any of the object-oriented programming languages developed was that they all required a programmer to learn a completely new language. C++ got round this problem by extending the C language with object-oriented programming similar to how Objective C was designed. C, which was a widely used language in the commercial sector, allowed programmers to continue programming in a language and style they were familiar with and to adapt more naturally to the object-oriented programming style. Through, among other things, better marketing, C++ succeeded where Objective C failed.

Next we will look in more detail at the state of object-oriented programming today.

2.1.2 The Current State of Object-Oriented Programming

A new object-oriented programming language that is rapidly growing in popularity is C# [HWG03, Wil02]. C# is a strongly typed object-oriented programming language written by Microsoft for their Visual Studio package and makes use of their .NET platform [JSY03]. It is intended to improve on C++ and most especially Java in terms of simplicity and performance. C# was built with the benefit of hindsight of previous object-oriented languages particularly C++ and Java. In many respects its syntax is closer to Java's than C++. Some advantages C# has over Java is that it provides a more natural object-oriented syntax for accessing member fields of an object than Java does. Below is a simple example of a *Person* class in C#.

```
// person.cs
using System;
class Person
{
    private string myName ="N/A";
    private int myAge = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
           return myName;
        }
        set
        {
           myName = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
           return myAge;
        }
        set
        {
            myAge = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }

    public static void Main()
    {
        Console.WriteLine("Simple Properties");

        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated
        //with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);
    }
}
```

The two most popular object-oriented programming languages in use today are C++ (we looked at this briefly in the previous section) and Java (which we will look at in more detail in the next section).

C++ is currently the most widely used object-oriented programming language. 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 a very popular language particularly in its use for development of the UNIX operating system [AA86]. This means that any C code can be compiled by a C++ compiler. In 1985 C++ was released to universities with virtually no support. By 1987 the first C++ conference was held attended by 200 people rising to 600 in the next year. By the early 1990's the number of users was estimated to be over 500,000 which made C++ the fastest growing programming language in the world. Below is an example of a simple *Person* class in C++.

```
using namespace std;
#include <iostream>
#include <cstring>

class person
{
    public:

    char *name;
    int age;

    person (char *n = "no name", int a = 0)
    {
        name = new char[100];
        strcpy (name, n);
        age = a;
    }

    person (person &s) // The COPY CONSTRUCTOR
    {
        name = new char[100];
        strcpy (name, s.name);
        age = s.age;
    }

    ~person ()
    {
        delete [] name;
    }
};
```

```
int main ()
{
    person p;
    cout << p.name << ", age " << p.age << endl << endl;

    person k ("John", 56);
    cout << k.name << ", age " << k.age << endl << endl;

    p = k;
    cout << p.name << ", age " << p.age << endl << endl;

    p = person ("Bob", 10);
    cout << p.name << ", age " << p.age << endl << endl;

    return 0;
}
```

2.1.3 Java

The other language that is in large commercial use today is Java. As this is the language that we have chosen to model we will look at Java in more detail. Java was developed by Sun Microsystems [Sun05a]. Java itself began as a small project. It was originally built as an internal project within Sun. Frustrated by C++'s syntax (which they considered to be confusing), API, and tools, an engineer at Sun called Patrick Naughton started work on a new technology called the Stealth Project. He was joined by James Gosling and Mike Sheridan and the project was renamed the Green Project. Originally they planned to use C++ as the language for the new technology but decided against it as they felt C++ was a complicated language and often misused. They also considered the fact that C++ did not have any form of garbage collection to be a disadvantage. They required a language that supported multi-threading, distributed programming, and better security than C++ provided. Most important of all they needed a language that was portable and could be used on many different types of devices.

They decided in the end to create a completely new language which they named Oak. By 1992 they were able to demo the Oak programming language. In June and July of 1994 after a three day discussion the team decided to refocus efforts and attempt to provide support in Oak for the internet. Although still in its early days the team felt that the internet would be a big market in the future. Another important event of 1994 was the discovery that Oak was already a registered trademark forcing them to change the name to Java. In October 1994 Sun executives were given their first demo of Java and it was made available for download shortly after. Java was first made publicly available at the SunWorld conference on the 23rd May 1995. The Java language was given a further boost at the same conference by the announcement that the Netscape browser [NCC05] would support Java.

Over the years there have been many releases of Java. However three of these releases are considered the most important, Java 1.0 (the original release) and Java 1.2 (also known as Java 2) and Java 5.0. Java 5.0 [Sun05h] was released during 2004 when the majority of the work in this thesis had been completed hence we will not be examining this version any further.

Java 1.0 lacked many of the key features associated with Java today. In particular its classes for handling graphics and GUI interfaces were basic and limited. In 1997 several important new features such as inner classes were added to Java with the release of Java 1.1. Java's next significant release came in 1998 with the release of Java 1.2 with major changes to the API where important new features such as Reflection [FF04] were introduced. Also Java's graphic and GUI classes were greatly expanded upon with the introduction of the Swing API [GRV03]. In 2000 Java 1.3 was released which contained only minor changes and bug fixes. In 2002 Java 1.4 [AGH00] was released. Again this contained only minor changes. As of 2004 it was the most widely used version of Java and it is this version which we have chosen to model in this thesis. Sun have segmented their API's into three platforms each targeting different programming environments.

- Micro Edition. This is aimed at environments with limited resources.
- Enterprise Edition. This is aimed at large distributed or internet environments
- Standard Edition [Sun05g]. This is aimed at workstation environments. This is the environment we have chosen to base our model on, although our work could be adapted and extended to other environments.

The following is a simple example of a *Person* class in Java.

```
public class Person {

  public String name;
  public int age;
  public Person(String name,int age){
    this.age=age;
    this.name=name;
  }

  public Person(){

  }

  public String toString(){
    return this.name + " " + this.age;
  }

  public static void main(){
    Person p = new Person("Justin",25);
    Person p2 = new Person("James",21);
    System.out.println(p.name + " " + p.age);
    System.out.println(p2.name + " " + p.age);
    System.out.println(p);
    System.out.println(p2);
  }
}
```

Java has several advantages over other programming languages. Its chief advantage and the one Sun Microsystems identifies as its core value is the "Write once, run anywhere" [Pol97] principle. A Java program can be written on any platform and can then be run on any platform that has a Java Virtual Machine written for it allowing programmers to write code that is truly platform independent. Another key feature that Sun sells Java on is Java's security. Users can download untrusted code over a network and have it run in a secure environment where it cannot spread viruses and cannot read or write files from the hard drive. Also Java's security model is highly configurable allowing different security levels and options to be set for different Java code. This security model is not limited to internet applications (Java Applets [Gos96]) but can be used with any Java code. Finally Java was designed with particular emphasis on network computing so it provides many classes that allow programmers to write powerful network and internet programs. One thing Java does not support is Multiple Inheritance, making use of only Single Inheritance. However many software engineers now view

Multiple Inheritance as bad practice. Also as Java is compiled into Java Byte Code and then interpreted by a Java Virtual Machine [LY99] it can be slower and more resource heavy than other languages, however as the speed and power of computers have increased over time along with the continued optimisation of the Java Virtual Machine this problem has practically disappeared. One of the most important optimisation was the introduction of Just-In-Time compilers (JIT). As of Java 2 a JIT compiler was added to Java's collection of tools [Sun05b]. JIT compilers work by compiling the Java byte code into machine instruction code. This is only done when the code is first encountered thus improving runtime performance by preventing Java byte code used multiple times having to be continually translated into machine code. The translated code is stored in a runtime cache. The effectiveness of a JIT compiler is dependent on how intelligent its choice of which byte code to precompile into machine code is. Some code will take longer to compile than it would to simply run the Java byte code especially if only used once.

2.2 Algebraic Specification

In this section we will briefly introduce *Algebraic Specification.* We will look in particular at *Many-Sorted* and *Order-Sorted* algebra. We will also examine how many-sorted algebra and order-sorted algebra can be considered to be equivalent under certain conditions. Finally we will briefly look at some of the important areas in which algebraic specification has been successfully applied. Algebraic specification is a complex subject, and we will not discuss it any detail. Useful reading on many-sorted algebras can be found in [ST99, GTWW77, GTW78, Wag81, MG85, EM85, Wec91, MT92, ABBK99]. Useful reading on order-sorted algebras can be found in [Gog78, GM92, GD94]

2.2.1 Introduction to Many-Sorted Signatures, Algebras, and Specifications

We wish to introduce three concepts. We will be following closely the work of [TS06] when introducing these concepts. The first concept we will introduce is that of a *Many-Sorted Signature.* A signature defines purely the syntax of a system. It consists of sort, operation, and constant names. It is purely a syntactic structure and does not appeal to a reader's understanding of what any symbols defined in the signature mean.

For example a signature might consist of an operation called *succ.* Although we might intuitively assume that *succ* calculates the successor of a given number there is nothing in the signature to define this to be the case. In the signature *succ* is declared purely as syntax and the semantics of it are not defined here.

The next concept we will introduce is a *Many-Sorted Algebra.* In brief, an algebra consists of sets of data with functions on the sets of data. The functions provide some basic tests and operations for working with the data. A many-sorted algebra can contain multiple sets of data and functions that can operate over different combinations of the data sets.

An algebra assumes an understanding of the semantics of its operations on the user's part. For example, suppose we have an algebra with a function as follows.

$$\_ + \_ : Nat \times Nat \longrightarrow Nat$$

Taking *Nat* to be the data set of natural numbers ($\{0, 1, 2, 3, \ldots, N\}$) and the underscorses (_) to represents the function's inputs, then intuitively we can assume that the function takes in two natural numbers, adds them together and returns the new number as the result. Therefore the algebra assumes an understanding of the semantics on the user's part (in the example it assumes and understanding of the + function).

In order to specify a system we will want to define the behaviour of a signature that does not require the user to have a pre-understanding of the semantics involved. To do this we will introduce our third concept, a *Many-Sorted Specification.* A many-sorted specification consists of the following:

$$Signature + Axioms$$

In our model the axioms will consist of *equations* and *conditional equations.* This again like the signature is syntactic and does not require the user to have a prior understanding of any semantic meaning.

A signature can be interpreted by many different algebras. Many of these interpretations might be considered to be invalid depending on a user's point of view. For instance an algebra of the booleans could map both *true* and *false* to a singleton value such as *1.* Although this would be a legal interpretation of the signature for the booleans, in itself it is not a very useful interpretation (although some researchers might feel that this algebra is just as important and valid as other algebras). We solve this by using the *initial model* (although it should be pointed out that other researchers with different view points may choose a different model).

Given algebras A and B over the same signature, we define a *homomorphism, h* as follows:

$$h : A \rightarrow B$$

Where for every C_A interpretation in A of a constant in the signature and C_B interpretation in B of a constant in the signature:

$$h(C_A) = C_b$$

and for x, y, and z variables in A, f_A operation in A, and f_B operation in B:

$$h(f_A(x, y, z)) = f_B(h(x), h(y), h(z))$$

A is initial if there exists exactly one homomorphism from A into any B in S, the class of all algebras.

2.2.2 Mathematical Preliminaries

In this section we define the mathematical notations that we will be using for *signatures* and *algebras.*

- A family of *sort* sets indexed by S called A is written as follows $(A_s \mid s \in S)$. For example S could be the set $\{nat, int\}$. A_{nat} could be the set consisting of $\{0, 1, 2, 3, \ldots\}$ and A_{int} could be the set consisting of $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$[1]. The set S is therefore simply the set of names of sorts and A_s is the set that represents the sort called s.

- We define S^* to denote the set of all finite sequences over S. This includes the empty sequence λ. S^+ is used to denote the set of all finite sequences over S minus the empty sequence. It is defined as follows $S^+ = S^* - \{\lambda\}$.

- We define the following

  $$A^\omega = A_{s_1} \times \ldots \times A_{s_n}$$

  for each $\omega = s_1 \ldots s_n \in S^+$.

- Finally we define operations as maps of the form $f : A^\omega \rightarrow A_s$ for some $\omega \in S^*$ and $s \in S$. When $\omega = \lambda$, that is an operation has no argument sorts, then the operation would be a constant $c \in A_s$.

[1]Of course, there are many other ways to represent A_{nat} and A_{int}.

2.2.3 Many-Sorted Signatures

A Many-Sorted Signature Σ is represented by the following triple.

$$\Sigma = (Name, S, < \Sigma_{\omega,s} \mid \omega \in S^*, s \in S >)$$

Σ consists of the following:

- The name *Name* of the signature, also called the identifier.
- A set S of sort names.
- An $S^* \times S$ indexed family of sets as follows:

  $$< \Sigma_{\omega,s} \mid \omega \in S^*, s \in S >$$

  The elements of this family are the *constant* (where $\omega = \lambda$) and *operation* symbols.

The following is an example of a many sorted signature $\Sigma_{NatBool}$.

$$S = \{nat, bool\}$$

$$\Sigma_{\lambda,nat} = \{0\}$$
$$\Sigma_{\lambda,bool} = \{true, false\}$$

$$\Sigma_{nat,nat} = \{succ\}$$
$$\Sigma_{bool,bool} = \{not\}$$
$$\Sigma_{bool\ bool,bool} = \{and\}$$
$$\Sigma_{nat\ nat,bool} = \{=\}$$

2.2.4 Many-Sorted Algebras

We now define Many Sorted Algebras of a signature $\Sigma = (Name, S, < \Sigma_{\omega,s} \mid \omega \in S^*, s \in S >)$. An algebra A consists of the following.

- An S-indexed family of sets

  $$< A_s \mid s \in S >$$

  Where A_s for each $s \in S$ is called the *carrier* of sort s and interprets the sort name s.

- An $S^* \times S$-indexed family of sets of constants and functions.

  $$< \Sigma^A_{\omega,s} \mid \omega \in S^*, s \in S >$$

  For each sort name $s \in S$ and each word $\omega = s(1)\ldots s(n) \in S^+$

  $$\Sigma^A_{\omega,s} = \{f^A \mid f \in \Sigma_{\omega,s}\}$$

  Where for each operation name there is a function of the appropriate type

  $$f^A \ : \ A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_s$$

  or

  $$f^A \ : \ A^\omega \rightarrow A_s$$

  which has the domain $A^\omega = A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_s$.

  Constants are defined as function with no arguments where for each sort $s \in S$ and empty string $\lambda \in S^*$

  $$\Sigma^A_{\omega,s} = \{c^A \mid c \in \Sigma_{\lambda,s}\}$$

  Where the element

  $$c^A \in A_s$$

  is called a constant of sort $s \in S$. This interprets the constant symbol $c \in \Sigma_{\lambda,s}$

An example of a many sorted algebra $A_{NatBool}$ that interprets the many sorted signature $\Sigma_{NatBool}$ as defined earlier is as follows:

$$A_{NatBool} = \{\mathbf{N}, \mathbf{B}\}$$

$$\begin{aligned}
&0^{\mathbf{A}} :\rightarrow \mathbf{N}\\
&true^{\mathbf{A}} :\rightarrow \mathbf{B}\\
&false^{\mathbf{A}} :\rightarrow \mathbf{B}
\end{aligned}$$

$$\begin{aligned}
&succ^{\mathbf{A}} : \mathbf{N} \rightarrow \mathbf{B}\\
&not^{\mathbf{A}} : \mathbf{B} \rightarrow \mathbf{B}\\
&and^{\mathbf{A}} : \mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B}\\
&=^{\mathbf{A}}: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}
\end{aligned}$$

2.2.5 Many-Sorted Specifications

A *many-sorted specification* consists of a signature plus axioms. A specification is purely syntactic and unlike algebras, it does not require a prior knowledge of the behaviour of the functions defined in it to understand how a specification defines a system. In our work we make heavy use of specifications in order to model object oriented languages. The axioms we use in specifications consist of equations and conditional equations (although other types of axioms could be used if desired).

We will not be continuing with the style of notation that we used for signatures and algebras when it comes to specifications. The syntax we used for these is not very convenient and is not very human readable. As part of the aim this thesis is to provide a way of formally specifying object-oriented programs in a way that is clear and understandable to an object-oriented programmer, for the rest of this chapter we will adopt a different style of syntax for specifications. This new syntax style is deliberately designed to be similar to object-oriented programming code. Also it should be noted that like the algebras, the new syntax does not contain constants. This is because we have treated constants as operations with no arguments and thus they are treated as operations. The constants therefore are now of the form $c :\rightarrow A$. The syntax for specifications are as follows:

```
$Specification\ Name\{$

  $Sorts\ \{A_1, \ldots, A_s\}$

  $Operations\ \{$
    $f_1 : A_i \times \ldots \times A_j \rightarrow A_k,$
    $\ldots,$
    $f_n : A_t \times \ldots \times A_u \rightarrow A_v$
  $\}$
  $Equations\ \{$
    $\ldots$
  $\}$

$\}$
```

It should be noted that although the *Name* part of the syntax is allowed to be whatever a user wishes, we often use the names of the sorts that it uses as the name of the specifications. Note we will discuss the format of equations and conditional equations in chapter 3. If we consider an algebra *Bool* it would be presented in the new syntax as follows.

$$
\begin{array}{l}
Specification\ Bool\ \{ \\
\quad Sorts\ \{B\} \\
\quad Operations\ \{ \\
\qquad true :\rightarrow B, \\
\qquad false :\rightarrow B, \\
\qquad succ : N \rightarrow N, \\
\qquad not : B \rightarrow B, \\
\qquad and : B \times B \rightarrow B, \\
\quad \} \\
\quad Equations\ \{ \\
\qquad not(false) = true \\
\qquad not(true) = false \\
\qquad and(true, true) = true \\
\qquad and(true, false) = false \\
\qquad and(false, true) = false \\
\qquad and(false, false) = false \\
\quad \} \\
\}
\end{array}
$$

As can be seen in the above example constants $0, true, false$ are all categorised as operations.

2.2.6 Order-Sorted Signatures, Algebras, and Specifications

Order-Sorted signatures, algebras, and specifications further extend the concept of the many-sorted varieties by imposing a partial order on the sort sets. It does this using the notion of a *subsort* which defines an ordering on sorts.

If we have two sorts s_1 and s_2 then we can say $s_2 \leq s_1$ which means that s_2 is a subsort of s_1. We introduce Order-Sorting in order to provide a way of expressing the inheritance relationship between Java classes in our model. As will be seen in chapter 4 the subsort predicate is used to indicate that one class is a subclass of another.

Adapting our notation for many-sorted specifications, we define order-sorted specifications as follows:

$$
\begin{array}{l}
\textit{Specification}\ A_1 \ldots A_S\ \{ \\
\quad \textit{Sorts}\ \{A_1, \ldots, A_s\} \\
\quad \textit{Subsorts}\ \{A_i \leq A_j, \ldots, A_m \leq A_n\} \\
\quad \textit{Operations}\ \{ \\
\qquad f_1 : A_i \times \ldots \times A_j \rightarrow A_k, \\
\qquad \ldots, \\
\qquad f_n : A_t \times \ldots \times A_u \rightarrow A_v \\
\quad \} \\
\quad \textit{Equations}\ \{ \\
\qquad \ldots \\
\quad \} \\
\}
\end{array}
$$

Consider the following specification where we combine the I and N integer and natural number specifications together. We have four operators, a successor operator *succ* that finds the next successive value of an integer, an integer predecessor function *predint* which calculates the predecessor of an integer number and returns it as an integer, a natural number predecessor function *prednat* which calculates the predecessor of a natural number and return it as a natural number value. Its behaviour is informally the obvious behaviour for a predecessor function for numbers greater than 0. As natural numbers do not have negative values then the for the value 0 *prednat* can be assumed to return 0. Finally we have 0, a constant operator (the equations have been omitted).

```
Specification IN {

  Sorts {I, N}

  Subsorts {N ≤ I}

  Operations {
    0 :→ N,
    succ : I → I,
    predint : I →
    prednat : N → N
  }

}
```

The *succ* and *predint* operators works on integer numbers. However as the specification defines natural numbers to be a subsort of the integers, the *succ* and *predint* operators can be applied to natural numbers also. However this is not the case for the *prednat* operator. This operates on natural numbers and although we have defined natural numbers to be subsort of integers this does not mean it is necessarily true that integers are subsort of natural numbers (they are obviously not). Therefore the *prednat* function cannot operate on integer numbers. As there are no negative values in the natural numbers it is therefore not true that every integer value has an equivalent natural number value. Therefore the *prednat* functions is restricted to natural numbers and we are not able to pass integer values to the function. The function *prednat* also behaves differently when passed the value 0 than *predint* does. Whereas *predint* would be expected to return the -1 integer value if passed 0 as its input, *prednat* is unable to do this as it can't return negative values so it is specified to return 0. Informally the equation definition for *prednat* would be as follows:

```
eq prednat(0) = 0
ceq prednat(N) = N - 1 if N > 0
```

2.2.7 Equating Order-Sorted Algebras to Many-Sorted Algebras

The work in this thesis makes use of order-sorted algebra to model the behaviour of class inheritance in particular with the use of the *subsort* relationship. Order-sorted algebra is considered by some to present problems that many-sorted algebras do not [Tuc06]. Specifically it is the view with some people that certain operators over order-sorted algebras would return results where, due to the order-sorting, it is unclear what the exact return type of the operator is. We will not be drawing conclusions on these viewpoints in this thesis. However in this section we will show how under certain conditions order-sorted algebra is equivalent to many-sorted algebra in order to provide an alternative approach for those who consider the results of order-sorted algebra operators to be ambiguous.

One of the requirements to allow us to be able to equate our order-sorted algebra to a many-sorted one is that our model needs a least sort. That is there must be a sort type for which all Java types are ultimately subsorts of. In Java, all classes ultimately inherit from a class called `Object`. Classes that are not specifically designated as inheriting from another class are considered to implicitly inherit from the `Object` class. In our model we often omit this implicit inheritance in order to keep the discussion and examples in this thesis simple and clear.

However, although the `Object` class can be considered the least sort with respect to Java classes, it does not cover simple types. In Java, simple types such as integers and booleans are not Java classes and therefore do not inherit from the `Object` class. Therefore our model would require a new sort type which would be a least sort for both `Object` and the simple data types. The `Object` class and simple date types would all inherit directly from this overall least sort. Our model does not incorporate this overall least sort as we are not concerned in the details of allowing our order-sorted model to be translated to a many-sorted one. We only discuss how it can be done in this section as an example of how our model could be adapted to use a many-sorted specification by researchers who prefer many-sorted specifications to order-sorted specification.

This idea was first presented in [GM92], see also [TM94]. An order-sorted algebra A is isomorphic with some many-sorted algebra B if A is *regular*: that is, there exists a *least* sort that all other sorts are related to by the subsort predicate, and if B satisfies a set of many-sorted equations defined below.

Subsort relations are translated into standard equations. For every subsort relationship $s \leq s'$ in the order-sorted algebra A we view the subsort relationship as a mapping from s to s' in the form of *embedded functions* or *coercions* called $c_{s,s'}$ where

$$c_{s,s'} : s \longrightarrow s'$$

The following set of equations must be satisfied for every subsort mapping $c_{s,s'}$ for the many-sorted algebra B

- The Identity equation. $c_{s,s}$ for each $s \in S$. That is every sort includes itself mapping from one sort to the same sort will always produce the same element.
- The Injectivity equation. $x = y$ if $c_{s,s'}(x) = c_{s,s'}(y)$ for each $s \leq s'$ in S. This makes sure that two different values in s do not map to the same value in s'
- The Transitivity equation. $c_{s',s''}(c_{s,s'}(x)) = c_{s,s''}(x)$ for each $s \leq s' \leq s''$ in S. This ensures that if a value a in s maps to a value b in s' and that value b in s' maps to a value c in s'' then a mapping of the value a from s to s'' will also map to the value c
- The Homomorphism equation.
  $c_{s,s'}(f_s(x_1, \ldots, x_n) = f_{s'}(c_{s,s'}(x_1), \ldots, c_{s,s'}(x_n))$ where $f_s : S^n \rightarrow S$ and $f_{s'} : S'^n \rightarrow S'$ are in A and $s \leq s'$ are in S. This states that all subsort relations must be homomorphisms as shown in the diagram below

$$\begin{tikzcd} S^n \arrow[r, "f"] \arrow[d, "{c_{s,s'}}^n"'] & S \arrow[d, "c_{s,s'}"'] \\ S'^n \arrow[r, "f_{s'}"'] & S' \end{tikzcd}$$

This means that for those who prefer many-sorted algebras that order-sorting can be considered to be a *notational convenience*, allowing us to easily specify inheritance between classes using subsorts which using the above equations can be adapted to a many sorted specification. As discussed earlier our work does not fully support the concept of a least sort. However it would be straight-forward to implement the *Object* class in our model and future work will need to be done to add this class correctly to if was desired that our algebra is regular and thus can be equated to many-sorted algebras. As this thesis is aimed at demonstrating modelling a wide range of Java functionality we have not focussed on the small details of ensuring our model has a least sort although we believe it would not be hard to introduce. The concept of a least sort and equating order-sorted algebras to many-sorted is only shown as a proof of concept as to how order-sorted notation can be used without moving away from many-sorted algebras, which as mentioned above is considered by some researchers to be more concise than order-sorted algebras.

2.2.8 Applications of Algebraic Specification

In this section we will look briefly at some of the important modelling tools that are available for writing algebraic specification and also at some of the key areas where algebraic specification has been utilised.

The first specification tool that we will examine is CASL [CAF05c, BM04, The04, CAF97] designed by CoFI (The Common Framework Initiative) [CAF05b] for algebraic specification and development. It is an expressive language and can be used to specify conventional software (requirements, design and architecture). The following example demonstrates a specification for numeric addition and successor in CASL. It is taken from the CASL website sample section [CAF05a].

```
spec Natural =
    free type Nat ::= 0 | suc(Nat )


spec Natural Order =
    Natural

    then free { pred < : Nat X Nat
    forall x , y : Nat

        . 0 < suc(x )
        . x < y -> suc(x ) < suc(y) }
end


spec Natural Arithmetic =

    Natural Order

    then ops 1 : Nat = suc(0);
    + : Nat X Nat -> Nat , assoc, comm, unit 0;
    forall x , y : Nat

     . x + suc(y) = suc(x + y)
end
```

The *Natural* and *Natural Order* define a natural number sort and successor operator *suc*. The *Natural Arithmetic* defines an addition operator on the natural numbers.

The language is not restricted to equational logic and allows other forms of logic such as *First Order Logic*. All of CASL's features exist in at least one other main existing algebraic specification framework. There are a few exceptions and the one most relevant to our work is that of subsorts. They have chosen to avoid imposing the condition of *regularity* on order-sorted algebras leaving the decision up to the person writing the specification. As we saw earlier it is important to the work in this thesis that we impose regularity on our model in order that we can equate order-sorted algebras to many-sorted algebras. CASL also allows *Operator Overloading*. CASL distinguishes between two different types of subsorts.

- The more traditional view of subsorts where one sort is considered to be a more specialised version of another sort (e.g. integers to real numbers)
- The domain of definition of a partial function (e.g. the even numbers for integer division by two)

Casl consists of three types of specification.

- *Basic Specification.* These declare signatures of sorts, total and partial functions, predicates, and subsorting relationships. Axioms are written in first order logic with equality and specify sort generation constraints.
- *Structured Specification.* These allow extensions of other specifications and introduce (among other constructs) translations, reduction, and union of specifications. They are formed by starting with Basic Specifications and combining specifications in various ways. These specifications do not impose any structure on the modules. They are used only to present specifications in a modular style.
- *Architectural Specification.* These allow the user to express specified software as being composed of separate units which can be reused. These units have clear interfaces so their usage is clear to the software engineer. These specifications allow a large specification to be represented in 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.

The next specification tool that we will examine is *Maude* [SRI05, CELM00, DM99, CDE$^+$01, CDE$^+$03, McC03, CDE$^+$04]. Maude is a high-level language that provides support for writing specifications that can then be executed using rewriting logic. It also supports declarative programming. Maude also provides support for equational specification as rewriting logic incorporates equational logic. The type of equational logic that Maude supports is *Membership Equational Logic.* This type of logic provides sorts, subsorts, and operator overloading. Maude also has metaprogramming capabilities including user definable module operations and the ability to declare strategies to control the deduction process of the Maude system. Maude is influenced by *OBJ3* (which we will examine later in this section) especially Maude's equational logic language which essentially contains OBJ3's equational logic language as a sublanguage. The following is a simple Maude example specification of natural numbers.

```
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
```

One of the main extensions that Maude makes on OBJ3 is that of membership equational logic [Mes98] which extends OBJ3's order-sorted equational logic. Membership logic is a specialised form of rewriting logic. A pair (T, R) is called a *rewrite pair* where T is a membership equational theory and R is a labelled collection of rewrite rules which are sometimes conditional. These rules will involve terms in the signature of T.

Maude uses *functional modules* (see the example of `fmod BASIC-NAT` above) to define theories in membership equational logic using equalites of the form $t = t'$ and membership assertions of the form $t : S$ where t is a term of sort S. The logic supports order-sorted algebra in the form of sorts, subsorts, polymorphic operator overloading, and the definition of partial functions. Functional modules are executed using rewriting techniques and operational semantics defined in [BJM00]. The functional module's equations are treated as rewrite rules and used until a canonical form is found. For this reason the equations should be terminating and sort decreasing in order to guarantee that equations will rewrite all terms to a canonical form. Typically the rewriting in a functional module will terminate with a single value as the outcome.

Maude's *system modules* strictly enforce that the rewrite rules $r : t \longrightarrow t'$ are *not* equations. From a computational view they are interpreted as *transition rules* with a system. From a logical view they are interpreted as *inference rules* in a logical system.

Unlike functional modules where the rewrite terms are expected to be terminating, that is not the case for *system modules* where the rewrite rules can be both divergent and infinite. In system modules, rewrite rules $t \longrightarrow t'$ are not treated as equalities but *local state transitions* where the left hand side is a pattern to be matched to the current state of the system and the right hand side is state that system can change to if it matches the left hand side. These state changes can run concurrently with other state changes so long as they do not overlap, thus rewriting logic is the logic of concurrent state change.

Maude supports reflection upon these modules through its meta level language defined in its *META-LEVEL* module. This allows a user to, among other things, define their own rewriting strategies. Also the *META-LEVEL* can be extended with new data types allowing the user to state formal requirements on parameters and the ability to change, initialise, and create new modules.

Due to Maude's use of rewriting logic, Maude is able to provide support for concurrent object-oriented computation. Maude provides a special syntax to support this in the form of *object-oriented modules.* Object modules in Maude can be used to model object-oriented systems which provides a more convenient syntax for modelling these systems than the system modules. An object module can declare a series of `CLASSES` each with a set of *attributes* which are the equivalent of fields in our model. When declaring instances of classes, the object modules require the user to explicitly declare values for all the attributes. Subsorts are used to define if one class inherits from another class. Methods are defined as messages to specified objects using rewrite rules. These rewrite rules are used to specify the behaviour of the system in relation to the messages that pass between the objects. Object modules are actually syntactic sugar that are internally turned into system modules.

We have chosen to model our system using Maude's functional modules. Although we could use the object modules we wish to be able to specifically model objects and classes using our own structure. There are two main reasons for this. The first and most important reason is that although we make use of Maude for our formal model, we do not wish to limit our model to only using Maude. If we were to use the object modules which are specific to Maude then it would become much harder to adapt our model to other specification languages. The second reason is that we want to explicitly define the complete structure of an object-oriented class. If we were to make use of the object modules then some of the internal workings of a class would be hidden. We feel it is important that a user can see the complete explicit structure of a class being modelled if they so wish. Also we wish to model using equational logic and not rewrite rules.

Maude itself is influenced by another specification language called *OBJ3* [GW88, GM96]. OBJ3 (and its predecessor *OBJ* [GWM+93]) is an algebraic programming and specification language. It is based upon the logic of order sorted equational logic algebra. It is syntactically similar to Maude which maintains many of *OBJ3*'s features and also adds an implementation of rewriting logic. OBJ3 is implemented in *Common Lisp*

Another older specification language is *Larch* [MIT05, GHG+93]. Larch specifications are written in two languages. The first language is defined for a specific programming language: these are called *Larch interface languages.* The second language is independent of any programming language and is called the *Larch Shared Language* (LSL).

The interface languages specify how program components communicate across an interface. It allows a user to define assertions about a program's state and also provides support for features such as side-effects, exception handlers, iterators and concurrency. The Larch specification language is used to write *auxiliary specifications* which are used by interface languages. Auxiliary specifications provide semantics for the primitive terms used in interface specifications. The principle behind this two-tier approach is to allow basic constructs to be written in the LSL level and for program language specific constructs to be written in the interface level. It is considered good practice in Larch to restrict most of the complexity of a specification to the LSL level. This is for several reasons.

- LSL specifications, being platform independent, will be more resusable

- Less mistakes will be made with LSL specifications as they are simpler than interface specifications as they lack platform related complexity
- The semantic properties of LSL specifications are easier to verify than the semantic properties of interface specifications

Although some Larch specifications can be executed most can not. Larch does allow tools to mechanically check assertions to verify specifications.

Algebraic Specification has been used in numerous areas. We will now present a range of examples (that is by no means exhaustive) that have employed algebraic specification.

- Specifying the semantics of Abstract Data Types (ADT's)[GH78]
- A set of mathematical tools for modelling microprocessors including pipelined and super-scalar models.[Har89, Har00, Har02]
- A strategy and tools for modelling and proving the compiling of a high level language to a low level language[Ste96]
- Creating Interface Definition Languages (IDL's) for the specificaiton of interfaces[Ree01, STR03]
- Formalising the overall structure as well as the structure of individual diagrams of *UML* using a Casl specification *UML* [RCA00]
- Using homomorphisms to model user defined and structured data types in imperative languages [ARZ99]
- Various approaches including algebraic modelling for specifying software systems concerned with concurrent systems [AMRW85, ABR99]
- Analysis of various approaches and requirements in modelling systems which are based on services[Fia02]
- Proving the correctness of the JavaCard achitecture bytecode verification security[BCDS02]
- Extending *Casl* to specify functional programs and to be able to perform rapid prototyping[SM02]
- Discovering classes of parametric hybrid systems that are decidable [Hen96, AR02]

- Initial work on establishing a framework for connecting together the various forms of order-sorted algebra [Ste02]
- Extending a type system to control access to state changing methods and prevent unanticipated changes to object references[Sko02]
- Methods for verifying the correctness of *Casl* Architectural Specifications [Hof03, Hof02]
- Using *CASL* to design 3D geometric modelling software [LAGB02, Duf97]
- The AGILE project which is concerned with modelling systems with mobile components [ABB$^+$03, Bau05]
- Establishing a formal relationship between different proposals for the semantical interpretation of Petri Nets [BMMS01, BBM03]
- Proving that states are behaviorally equivalent using Hidden Algebra which distinguishes between visible and hidden sorts for data [GM00, GLR03]
- Techniques for specifying and verifying the correctness of cryptographic protocols[HW03, BAN96]
- Structuring concepts for Rule-Based Systems that are independent of the types of rules and to what the rule are applied to[KK03]
- Using Multialgebra to combine multiple specifications in different algebraic frameworks[LW03, Lam02]

2.3 Object-Oriented Program Modelling

In this section we will look at the key areas in which algebraic specification has been applied to object-oriented programming languages with particular emphasis on Java. We will also look at other work besides algebraic specification which is relevant, and explain how our work will differ from these approaches.

The most relevant work in algebraic specification of object-oriented program is that of Henkel et al [Hen04, HD03, HD04a, HD04b]. They too are modelling Java classes algebraically but their work differs from ours in that the entire specification process is automated. Like us they use the Java Reflection API [McC98, Gre05, Sun05c] to discover the signature of a class but they also use an automated discovery tool to build the semantics automatically. Their tool takes in a Java class and uses the reflection API to discover the names of all the methods and constructors and also any input parameters and return types. Their tool then generates test terms based on this extracted signature to test the behaviour of the methods it has discovered and hence attempts to construct equations. The process itself is quite complex and works by analysing the results returned from tests and, if necessary, generates new test terms based on the results to test the class further and further refine the equations. Because this is automated the results can neither be considered to be complete or correct as it is possible that automated tool may not discover all possible results for the behaviour of methods or it may form incorrect conclusions based on the results it receives. They contribute to work previous done in [AH00] by creating a mechanism that allows the integration of algebraic rewriting techniques into Java, and they are more interested in the automation of the process than in the soundness and completeness of the specification itself. They themselves admit that the more complex the class being modelled, the more incomplete the discovered specification is likely to be. Our work differs from theirs as we are more interested in the completeness and correctness of the model itself rather than the automation. Unlike us they do not model fields, expecting them to be defined by public methods. Also their work is mostly limited to modelling container classes such as arrays and lists whereas we are interested in modelling a wide and varied range of Java classes and functionality. However we do feel that their tool, with adaption, could be used as a starting point for generating an initial set of semantic equations for a class. This would be especially useful for encouraging users who are less familiar with algebraic specification, allowing them to use the automatically generated equations to build upon to create a more concise, complete, and correct algebraic semantics for a class. However it should be pointed out that their approach does require an implementation of a class before it is able to generate equations for the semantics. Therefore it is limited to pre-existing Java class and could not be used in designing new classes.

Another important work in specifying object-oriented programs is that of the *Java Modelling Language (JML)* [JML05, LC05, LBR05, LPC+05] and consists of a variety of tools [BCC+wn] for specification writing, testing and debugging. JML follows Eiffel's method of using expressions in assertions and combines it with a model-based approach. As well as pre and post conditions it allows assertions to be intermixed in the Java code to aid in verification and debugging. The expressions use an extended form of Java's own expressions and add extra notation such as quantifiers. A simple example of a JML specification for a function that returns the integer square root of a number is as follows:

```
public class IntMathOps4 {
    /** Integer square root function.
     * @param y the number to take the root of
     * @return an integer approximating
     * the positive square root of y
     * <pre><jml>
     * public normal_behavior
     * requires y >= 0;
     * assignable \nothing;
     * ensures 0 <= \result
     * && \result * \result <= y
     * && y < ((\result + 1) * (\result + 1));
     * </jml></pre>
     **/
    public static int isqrt(int y)
    {
        return (int) Math.sqrt(y);
    }
}
```

The JML is embedded within the comments of the Java file (although it is possible to write JML specs in a seperate file and link to them from the main Java file). The above JML spec says that the input to the function must be greater than or equal to 0, that the method is not allowed to assign any values to fields (`assignable \nothing`), the final result multiplied by itself should be equal or less than the original value and that the final result plus one multiplied by the final result plus one should be greater than the original value. The JML specification can be used to create API specifications listing containing the formal JML specification as well as the informal textual specification. The above specification can be compiled into the Java program using a special JML tool to create design by contract conditions within the compiled Java program. It can also be used by other tools for testing, verification and debugging amongst other things. JML therefore provides a way of axiomatically specifying Java programs by embedding the JML specifications into Java comments which can then be extracted and manipulated using special JML tools.

Another key piece of work in object-oriented program specification is that of [Mül02, MPH97, MPH97]. They provide modular specification and verification techniques for a language called *Mojave* which is a subset of sequential Java with an added type system for alias control. Mojave is quite restrictive on many of full Java's abilities, omitting static overloading, user-defined constructors, abrupt completion (e.g *break,return*), exception handling and arrays among other things. Similar to that of JML, a Hoare Style logic of pre and post conditions are used on methods to specify the behaviour of the system. They also use a type system to not only give type declarations but also to provide alias information which is used to control sharing. They use an ownership model for their type system which takes the view that groups of objects which work closely together to complete a common task are *dynamic components*. Some of the objects within these components are used to interact with other dynamic components and are called *interface objects*. The other objects purely work internally to the component and are considered to be the internal *representation* of the component. The aim is to control references to the representation objects and thus prevent a method from causing unwanted side effects to references that need to be protected. One crucial problem with their techniques which they admit themselves is that they are too complex to be used by programmers.

In addition to these three key pieces of research into the specification of object-oriented programs we list a selection of other interesting works in the field.

- A Hoare logic for reasoning about Object-Oriented programs [AL97]
- Formal specification and verification techniques for object-oriented programs that use subtypes [LW90]
- Defining behavioral subtypes to allow modular reasoning for adding new subtypes to object-oriented programs [Dha97]
- A specification technique to use the inheritance of specifications to force appropriate behaviour on subtype objects [DL96]
- Using a tool called LOOPS to translate Java programs into a high order logic to be used by a theorem prover for reasoning [JvdBH+98]
- A formal notation for the specification and verification of software components in Java applications [CC99]
- A technique for specifying, refining, and proving properties of a small Java program using standard categorical constructs [Cla99]
- Extending Java to include assertions generated by Object-Z and CSP [Fis99]

- A performance model of Java execution using Petri Nets [RS00]
- An approach that adds access control to object references to limit the effects of aliasing while still allowing full referential object sharing [KT99]
- Defining the axiomatic semantics of a small Object-Oriented langauge called *Ecstatic* [RL96]
- Discussion on extending Java to check Object Invariants [RLS97a]
- Introducing the concept of Virginity which provides a way of specifying that an object is not globally reachable and can therefore be used in the implementation of a higher level of abstraction [RLS97b]
- A series of tools that allow an algebraic specification of an Abstract Data Type to be used in unit testing of object-oriented programs [DF94]
- A tool called Bandera that can automatically extract a Finite-State Model from Java source code which can then be used by a variety of verification tools [CDH+00]
- A tool for algebraically testing object-oriented programs that use side-effects to implement Abstract Data Types [HS96]
- A tool called iContract that adds Design By Contract features to Java code [Kra98]
- An experimental tool based on an algebraic continuation passing style (CPS) semantics for the verification of properties of a sequential imperative subset of the Java language [SM06].
- Using a coalgebraic approach to model Object-Oriented classes and objects and extending the approach to model inheritance [Jac96b, Jac96a, JP03].

Chapter 3

From Java Classes To Algebraic Class Specifications

In this chapter we will examine the process of modelling a *Java class* as an *Algebraic Class Specification* (ACS). The aim of this chapter is to show how to translate a standard Java class into the formal specification that we will define in this chapter. The ACS is not a *full algebraic specification* (FAS) but a simplified version of the FAS. The FAS as we will see in Chapter 4 can be quite lengthy and complex as it has many equations and operators used to define the internal working of the class structure. Our ACS is designed to be a more human readable way of writing down a class' algebraic specification. As we will see in this chapter, it contains all the information that we feel would make it clear to a reader how a class behaves, particulary its methods and constructors. The ACS can then be used to build the FAS, as will be seen in Chapter 4. We will look at the complexities that arise from defining a Java class as an ACS and how these problems are overcome. We will also look at how modelling a Java class as an ACS extends and builds upon the work on modelling interfaces by [STR03].

We are interested in specifying the *public interface* of a class. That is, all the *fields*, *methods* and *constructors* that are declared as being **public**. We are interested in modelling the behaviour of fields, constructors and methods. We wish to be able to model methods that can both return a value and change the state of a class instance. We are also interested in modelling the behaviour of *class inheritance* (although most of the specification of inheritance behaviour is not discussed until Chapter 4). Later in Chapter 5 we will look at how we further the model by introducing specification techniques for arrays and reflection classes. A grammar for the syntax of Java itself can be found at [Sun05d]. There are few aspects of Java programming that at present we cannot model but would like to do so in the future. The following are the two key concepts that we are not able to model at present.

- *Exceptions.* Exceptions in Java provide a useful mechanism for runtime error control. At present we are unable to model Java Exception classes however we believe that with further investigation and work it would be possible to incorporate this into our model. See Section 6.7.1 for a more detailed discussion on the subject.

- *Static Methods.* Static methods in Java are methods that can be executed that do not require a valid class instance to execute them on. Although we cannot model these at present we believe that it would not be too difficult to incorporate them into our model and as such have created a section in the ACS where these can appear in future work.

Section 3.1, will identify what we consider to be the core aspects of *object-oriented classes*. This will include the concepts of a *Class*, its *methods*, *constructors*, and *fields*, and the principles of *inheritance*. At this stage the concept of inheritance is kept relatively simple, where a class inherited by another class is defined in the ACS as a name link to the inherited class. The modelling of inheritance requires many equations and operators to fully define how a class inherits from another class and we will concentrate on modelling the actual inheritance issues in classes in chapter 4. The aim in section 3.1 will be to identify and define the basic features of a class that we wish to model. In section 3.2 we look at the notation we have designed for writing ACSs. This will look at how we model each of the concepts we discussed in section 3.1. As will be seen, we have attempted in our notation to make it close syntactically to Java in order to provide a more natural and readable specification for a Java programmer. In section 3.3 we will show how a set of Java class examples are modelled by ACSs. We will use examples to illustrate the process of modelling the concepts described in section 3.1 using the concepts discussed in section 3.2. This will be split into two stages. First we will transform a Java class into an algebraic interface which purely defines the syntax of the class. We will then take the interface and add equations to it to create the ACS. Section 3.4 will then give an overview of the work done by [STR03] on modelling interfaces and signatures. Also in section 3.4 we will relate the work of [STR03] to our work and show how our ACSs relate to interfaces and signatures. We will show that our work goes into more detail than the work they do relating to the concept of the *body* of the interface. Finally in section 3.5 we will examine part of the program we have implemented for building an ACS from a Java class.

3.1 Object-Oriented Classes

In order to attempt to model Java classes as ACSs we need to identify the key components of a class in object-oriented programs [CN91]. It is beyond the scope of this thesis to model a complete representation of all the concepts of an object-oriented class seen in the very wide range of object-oriented programming languages. However we will define a modelling framework for what we consider to be the core features. Some other additional features that we have not been able to implement in our modelling framework are discussed in section 6.7 where we will give suggestions on how they could be incorporated into our model. Throughout the remainder of section 3.1 we will use excerpts from a Java class called `Person` to illustrate each concept.

```
public class Person{

    public String name;
    public int age;

    public Person(){

    }

    public Person(String aName,int anAge){

        name=aName;
        age=anAge;
    }

        public void addYear(){

        age=age+1;

    }

    public String toString(){

        return name + " Age:" + age;

    }
}
```

3.1.1 Class Name

The first component of a class is the *classname.* A classname is a unique name which cannot be shared with any other class. In Java, classes are stored in *packages* which are themselves given a name. In Java, a classname is constructed in the following format:

$$name1.name2.name3. \ldots .classname$$

This is part of a package structure where `name1`, `name2` and `name3` are packages. Although from this it can be read that `name1` contains `name2` (and similarly for `name2` and `name3`), `name2` is *not* a subpackage of `name1` but are completely separate packages. This therefore means that `name1` and `name1.name2` should be treated as entirely separate packages (although often in Java there is an inferred relation between them).

The package and classname structure, shown above, is shortened to *classname* without the package by creating linking references to the corresponding package name using the *package* statement followed by the package structure. Other packages can also be made easier to access by the use of the *import* statement. For example to use a class from another package to create a *class instance* (also called *objects*) variable, you would have to write the following:

```
mypackage.MyObject myvar;
```

This can be shortened to:

```
MyObject myvar;
```

by declaring an import statement at the start of the class file:

```
import mypackage;
```

This is not to be confused with the *include* statement from C which copies code from a named file. Here is an example of a classname in Java.

```
public class Person{
```

where `Person` is the classname itself.

3.1.2 Fields

Fields are variables which belong to class instances. In the class Person there are two fields. One is a String field called name and the other is a primitive data type *integer* field called age. The intuition being that a person has a name and an age.

```
    public String name;
    public int age;
```

The primitive data type *int* is not a class instance. It is one of several basic data types built into Java. They are not classes largely for efficiency and programming convenience. It should be noted that String is a class itself and hence name is a class instance of String. It has methods, fields and constructors. Its main purpose is to represent strings in Java. With class instances, Java uses *dynamic binding.* With dynamic binding the field representing the class instance is bound to the corresponding class definition at runtime. This means that a user can replace the original class definition with a new one while a program is running. So long as the new definition has the same signature as the old definition then the program will be able to continue to run uninterrupted but will exhibit different behaviour in relation to the newly added class definition. Our modelling technique is unable to handle dynamic binding. Each class is specified before it is executed for testing purposes. Therefore if someone was to replace the original class definition with a new one, then the specification would not match the new definition. In order to do so the new class would need to be respecified. It should be noted that this is not a limitation of the model itself but is purely down to the fact that we choose to fully specify and create the executable model before we test it. The model could be adapted to cope with dynamic binding by having it specify each class every time it is called. This would ensure that the model would always be using the most recent behaviour of the class. In order to maintain clarity and also as the practice of substituting new classes at runtime would be considered an unusual occurrence we have chosen to stay with the process of fully specifying classes before testing them.

3.1.3 Constructors

Constructors are used to create class instances. In Java, the name of the constructor is always the same as the classname. A class can have more than one constructor provided their *signatures* are different. That is, they do not have the same number and types of arguments. This is called *operator overloading.* 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.

In the example below there are two constructors. The first constructor has no inputs and when called it simply creates an empty class instance. Note, in this case the empty no-argument constructor must be implicitly included if we wish it to be present because we have also included another constructor. However, if we had no constructors at all defined, then the default is to automatically assure the existence of the basic no-argument constructor.

The second constructor is called with `String` and integer arguments. These are used to initialise the values of the fields `name` and `age` (see section 3.1.2).

```
public Person(){

}

public Person(String aName,int anAge){

    name=aName;
    age=anAge;
}
```

3.1.4 Methods

In Java, *methods* take on the role of procedures and functions in procedural programming languages. Methods need not return a value. If they do not return any value their return type is *void.*

In the `Person` example below there are two methods. In this case neither method take arguments. However it is common for methods to take in arguments. The first method is `addYear` which simply increases the value of the `age` field by one. The `addYear` method does not return a value so its return type is *void.* The second function, called `toString`, builds and returns a `String`. This string is formed by concatenating the two fields and a string literal to create a human readable sentence describing the `Person` based on the values contained in the fields.

```
    public void addYear(){

        age=age+1;

    }

    public String toString(){

        return name + " Age:" + age;

    }
```

3.1.5 Inheritance

Inheritance is the process of *extending* and *specialising* a class [CN91]. For example, a new class could extend an existing class by adding new fields, methods and constructors. It could also modify existing methods and constructors. The new class is a *subclass* of the inherited class. In Java, if the new class contains a method with the same signature (same name, input types and return type) as the inherited class then the old method is *overridden* by the new method. Note, in other object-oriented languages this is more complex, for example in C# [HWG03].

We will define a new class `Student` that inherits from `Person`. In Java, the keyword *extends* is used to denote inheritance. This means that all the fields and methods from `Person` are also available to the `Student` class. `Student` has a new integer field called `studentId` in which a personal identification number for a student can be stored. `Student`'s constructor initialises this value along with the `name` and `age` fields.

The other important thing to note about this class is the redefinition of the `toString` method. Because this already exists in the inherited class `Person`, when we define it in `Student` we *override* it with our new definition. However we can still access the original method of the `Person` class by use of the keyword *super* which allows us to access overridden methods from the inherited class. In the `Student` example, this allows us to use the `Person` class' original `toString` method to build a new `String` with `studentId` concatenated onto the end.

```
public class Student extends Person{
    public int studentId;

    public Student(String aName, int anAge, int aNum){

        name=aName;
        age=anAge;
        studentId=aNum;

    }

    public String toString(){

        return super.toString() + studentId;

    }

}
```

3.2 Algebraic Class Specifications

In this section we will look at the notation that we have developed to model object-oriented classes. There are two parts to this notation. The first is the algebraic class interface which models purely the class syntax without any semantics. For the second part we add variables and equations (axioms) to define the semantics of the class and thus create an Algebraic Class Specification (ACS). We will also examine how we model each of the individual core aspects of an object-oriented class as identified in section 3.1. It should be noted that as we are concentrating on modelling Java and ultimately creating an FAS in Maude, the syntax we have created shares similarities with that of Java and Maude.

3.2.1 Concrete and Abstract Syntax

Throughout this section we will be introducing new syntax for each of the object-oriented concepts in the Java language that we are going to model in our ACS language. As the ACS will eventually be translated into an FAS written in Maude, the ACS syntax is essentially that of Maude with some added syntactic sugar to provide additional clarity and simplification.

For most of this chapter we will be using a concrete syntax to define these concepts in our model. However, later, for the more complex constructs such as an Algebraic Class Interface and an Algebraic Class Specification which are built from the earlier syntactic concepts, as well as concrete syntax we will be making use of an abstract syntax. This abstract syntax is used to combine the simpler syntactic concepts into the more complex syntactic components. The abstract syntax consists of a series of operators and equational definitions for the operators.

For the simpler syntactic concepts we will define each concept as follows.

1. A cartesian product of the Java concept we are modelling which will be used to store all the relevant information for that concept.
2. A general version of the actual concrete syntax that will appear in an ACS.
3. An operator that allows us to write the actual concrete syntax as shown in the general case.
4. An actual specific example of the syntax concept that has been introduced.

Where we differ from this format and introduce abstract syntax we will clearly identify in the text.

3.2.2 Identifier Names

Both classnames and inheritance classnames share the same sort type *Name*. Classnames are unique as discussed earlier. Identifiers must begin with either a letter, an underscore(_), or a Unicode currency symbol (e.g. $). The initial character can then be followed (optionally) by one or more letters, underscores or currency symbols.

We treat *Name* and *Sort* as the same types for notational convenience and hence we use them interchangeably. With *Sort* we make use of the ^{n} notation to denote a list of *Sorts*.

$$\begin{aligned}Sort^0 &= \lambda\\ Sort^1 &= Sort\\ Sort^{n+1} &= Sort^n \times Sort\end{aligned}$$

Where λ denotes the empty list of Sorts. To denote $Sort^n$ where n can be any integer value from 0 to infinity we use the notation $Sort^*$. To denote

$Sort^n$ where n can be any integer value greater than 0 we use the notation $Sort^+$.

For example $Sort^*$ denotes a sequence (or list) of zero or more $Sorts$ (eg. The sequence $sort1 \times sort2 \times sort3$ can be considered to be in $Sort^*$ where $sort1$, $sort2$, and $sort3 \in Sort$).

We omit the definitions for the * and $^+$ notation for other data types that we declare throughout the rest of this chapter as they can be considered to be similar to the definition above.

3.2.3 Fields

A field is a pair consisting of a name and a sort type. 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 a class instance of a certain class type. We define the following for fields.

$$Field = Name \times Sort$$

Fields are written in the following format.

$$fieldname \ : \ sorttype \ .$$

This will create a field of sort type *sorttype*. We define an operator to build field operators:

$$\_ : \_ \ : \ Name \times Sort \rightarrow Field$$

where $\_ : \_$ is the operator name. In applying the $\_ : \_$ operator, we replace the _'s with the *Name* and *Sort* arguments.

An example of the *age* field from `Person` is

```
age : Int .
```

3.2.4 Constructors

A constructor has a name (which is always the same as the name of the class being modelled) and (optionally) arguments. Each class constructor is distinguished by its input parameters. We define constructors as follows.

$$Constructor = Name \times Sort^*$$

Constructors are written in the following format.

$$ClassName\ :\ inputtype.$$

Where the list of sorts, *inputtypes*, could be empty. This will create a constructor of sort type *Constructor*. We now define a family of operators for building constructor operators:

$$\_ : \_^n\ :\ Name \times Sort^n \rightarrow Constructor$$

Where $\_^n$ is used to denote a sequence of n underscores. An example of an operator for a two argument constructor is as follows:

$$\_ : \_\ \_\ :\ Name \times Sort \times Sort \rightarrow Constructor$$

An example of a two argument constructor from `Person` is:

```
Person : String Int .
```

3.2.5 Methods

A method consists of a name, a list of sorts which are the method's input types and another sort which is the method's return type. If a method returns no value, then its return type is *void*.

Methods may either solely return information about a class instance (*Queries*), solely modify a class instance (*Commands*) or do both. For instance we may have an integer field called *number*. If this field is declared as being *private* then we will not be able to access it directly outside of the class as the field will not be part of the class' public interface (and hence the field itself would not appear in our model). In order to view and change *number* we would need to provide two methods. The first would be something similar to the following:

```
public int getNumber(){
    return number;
}
```

This method allows us to view the current value of *number*. It does not allow us to change the state of *number*. We would call this method a *query* or *accessor* method. The second method would be similar to the following:

```
public void setNumber(int i){
    number=i;
}
```

This method allows us to change the current state of *number*. We would call this method a *command* or *mutator* method. The query/command model would require that there be a clear divide between the two types of method.

In our model we have decided to allow methods that do both (and hence a method which might be traditionally viewed as a query is also capable of changing the state of the system as well) rather than adapting the query/command model. We will show later how we actually model the execution of methods that can both return a value and change the state of the system. Note that we still informally, on occasion, refer to queries and commands where it is convenient to distinguish different types of method.

We define methods as follows:

$$Method = Name \times Sort^{*} \times Sort$$

Methods are written in the following format.

$$op methodname\ :\ inputtypes\ \longrightarrow\ returntype\ .$$

This will create a method of sort type *Method.* We now define a family of operators for building method operators:

$$op\ \_ : \_^{n} \longrightarrow \_\ :\ Name \times\ Sort^{n} \times Sort \longrightarrow Method$$

Again we make use of $\_^{n}$ notation to denote a sequence of underscores. An example of a single input argument method operator is as follows:

$$op\ \_ : \_ \longrightarrow \_\ :\ Name \times\ Sort \times Sort \longrightarrow Method$$

An example of a single input argument method from `Person` is:

```
op addAge : Int -> Int .
```

For methods that do not return a value (i.e. `void` methods) we define the return type as being `void`. For example:

```
op ameth : Int -> void .
```

We will show later in Section 4.2.2 how we define a special sort type `void` to accommodate this. For now `void` should be viewed as another sort type and treated in the same way as any other sort used as a return type. As we would never define the equation for calculating the new class instance part of a `void` method, the `void` sort type is never evaluated to an actual value.

3.2.6 Operations

At present it is proposed that the operations will be the *static* methods of a class, however this is currently not implemented yet. For reasons why see section 6.7.

3.2.7 Interfaces

When modelling a class we first identify its public interface or signature. This defines the syntax of the class by stating which methods and fields are available together with their arguments and return types. However, there is no definition of the *semantics* of the class at this stage. In the class interface we wish to include the syntax for all the object-oriented aspects stated above.

An interface is a named collection of methods, constructors and fields that optionally *extends* (inherits from) another interface. We define interfaces as follows:

$$Name \times Name \times Field^* \times Constructor^* \times Method^* \times Operation^*$$

Interfaces are written in the following format:

Interface *name* [**Extends** *ename*]{
  Fields {*fields*}
  Constructors {*consts*}
  Methods {*meths*}
  Operations {*ops*}
}

Note that **Extends** is optional. If a class does not extend another class then we omit it. In Java, all classes ultimately inherit from the `Object` class. `Object` does not inherit from anything else, so, in modelling Java, the interface only omits the **Extends** part for the `Object` class (however in practice we usually omit the **Extends** clause for any classes that are inheriting directly from `Object` for covenience). Also note that we use *Bold* type face to denote text that exists verbatim within the structure. We define interfaces as being of sort type *Interface*. We also define an abstract syntax operator called *MakeInterface* which we use to build an interface from its component parts discussed earlier in this section.

$$\begin{aligned} MakeInterface \ &: \ Name \times Name \times Field^* \times \\ &Constructor^* \times Method^* \times Operation^* \ \longrightarrow \ Interface \end{aligned}$$

We define *MakeInterface* in terms of the concrete syntax using the following equation:

$$\begin{aligned} &MakeInterface(name, ename, fields, const, meths, ops) = \\ &\quad \textbf{Interface}\ name \\ &\quad \textbf{Extends}\ ename\{\ \textbf{Fields}\{fields\} \\ &\quad \textbf{Constructors}\{const\}\ \textbf{Methods}\{meths\} \\ &\quad \textbf{Operations}\{ops\}\ \} \end{aligned}$$

We define a special case of *MakeInterface* to deal with case that a class does not inherit from another class.

$$\begin{aligned} &MakeInterface(name, nil, fields, const, meths, ops) = \\ &\quad \textbf{Interface}\ name\ \{\ \textbf{Fields}\{fields\} \\ &\quad \textbf{Constructors}\{const\}\ \textbf{Methods}\{meths\} \\ &\quad \textbf{Operations}\{ops\}\ \} \end{aligned}$$

Where the word *nil* means *no class*, that is it is used to indicate that a class does not inherit from another class. When *nil* is passed in as the extending class name to *MakeInterface* then the **Extends** part of the interface is omitted.

We also define abstract syntax projection functions to project out each component of an interface.

We use *getName* to extract an interface's name:

$$getName : Interface \longrightarrow Name$$

This is defined by the following equation:

$$\begin{aligned} &getName(MakeInterface(name, ename, fields, const, meths, ops)) \\ &\quad = name \end{aligned}$$

Other projection functions for the other components of an interface are defined in the obvious way.

3.2.8 Algebraic Class Specifications Preliminaries

In order to be able to define the structure of an Algebraic Class Specification (ACS) we need to introduce three new concepts.

- Equations. These are used to define the behaviour (semantics) of a class, specifically the behaviour of the methods and constructors.
- Variables. These are used in equations. The specification language Maude requires all variables to be explicitly declared. As we are using Maude for our FAS specifications we will be providing a section in the ACS to explicitly declare all variables used.
- Hidden Operators. These are special operators that are not part of the interface specification of a class but are still neccessary in specifying the behaviour of a class and to aid in simplifying specification equations.

3.2.9 Equations

In order to be able to create an ACS we need to add a definition of the semantics of a class to a class interface. We do this using *equations*:

$$eq\, Term1 \;=\; Term2\;.$$

We can also have *conditional equations*:

$$ceq\, Term1 \;=\; Term2\; if\; Condition\;.$$

With conditional equations we only assert the equality of *Term1* and *Term2* if *Condition* is true. We use the standard Maude definition of a *Term* [CDE+04].

When invoked, a method potentially returns a pair consisting of a return value and a new state for a class instance. We need to differentiate between equations defining a method's return type (queries) and equations defining how a method changes the state of the system (commands). Given a method called *meth* for example we use the following notation for equations relating to the state change:

$$meth(inputs)_o$$

and the following notation for equations relating to *meth*'s return type.

$$meth(inputs)_q$$

For methods that do not return a value (`void` methods) we leave the behaviour of the $meth_q$ operator undefined. For methods that only return a value and do not change the state of a class instance we leave the behaviour of the $meth_o$ operator undefined. How we actually handle the use of the method notations as well as handling inheritance and field access in the FAS will be dealt with in chapter 4 when we look at how we expand an ACS to an FAS written in Maude.

It should be noted that in our model the the concrete syntax for assigning a value to a field looks like the following:

$$afield := avalue$$

This is different to the Java field assignment operator which is simply =. This has been changed to avoid confusion with the equation equals operator. Algebraically, the syntax for := is defined as follows:

$$\_ := \_ \ : \ Name \times Sort \to Sort.$$

Where the *Sort* input type is the sort type of the field called *Name*. Note that we are actually defining a set of := operators for each sort with an accessible field.

3.2.10 Variables

Variables are used in equations and every variable used has to be declared together with its sort type. We only need to define variables due to requirements of the Maude software which we use for the executable specification.

We define variables as:

$$\textbf{var}\ varname\ :\ Sort\ .$$

3.2.11 Hidden Operations

The final section we need to create for the ACS are *hidden operations*. These are used to define constructs such as constants which are used in the equation definitions. For instance in order to equationally define a stack we make use of a constant that represents an empty stack. This would be done by defining an operation *op EmptyStack* $: \to$ *Stack*. The *EmptyStack* operator is not an aspect of the original class that we are modelling but is considered in our specification to be necessary 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**. We also use this section to define operators that can be used to simplify the specification of a class. Hidden operators can be treated at present as the same format as *Operations* (which we define as the standard Maude definition for an operator [CDE+04]) hence the same type.

3.2.12 Class Specifications

To describe the behaviour of a class using its algebraic class interface we add semantics to the interface (using equations) to create the *Algebraic Class Specification* (ACS). At this point the ACS for the class is complete. We define the following for ACSs:

$$\begin{aligned} ClassSpec = & Name \times Name \times Operation^* \times Field^* \times \\ & Constructor^* \times Method^* \times Operation^* \times \\ & Variable^* \times Equation^* \end{aligned}$$

ACSs are written in the following format.

Class *name* [**Extends** *enames*]{
  Hidden{*hops*}
  Fields{*flds*}
  Constructors{*const*}
  Methods{*meths*}
  Operations{*ops*}
  Variables{*vars*}
  Equations{*eqs*}
}

As in section 3.2.7 the **Extends** is optional for the same reasons given in that section. Also, again we use **Bold** typeface to denote strings that exist verbatim within the definition. We define ACSs as the sort type *ClassSpec*. We define an abstract syntax operator called *MakeClass* to take in an interface and a class' semantics defined using variables and equations to build an ACS.

$$\begin{aligned} & MakeClass\ :\ Interface \times\ Operation^* \times Equation^* \times\ Variable^* \\ & \quad \rightarrow ClassSpec \end{aligned}$$

MakeClass is defined in terms of the concrete syntax as follows:

$$\begin{aligned}
&MakeClass(MakeInterface(name, ename, fields, const, meths, ops),\\
&\quad hops, eqlist, varis) =\\
&\textbf{Class}\ Name\\
&\textbf{Extends}\ ename\{\ \textbf{Hidden}\{hops\}\\
&\textbf{Fields}\{fields\}\ \textbf{Constructors}\{const\}\\
&\textbf{Methods}\{meths\}\ \textbf{Operations}\{ops\}\\
&\textbf{Variables}\{varis\}\ \textbf{Equations}\{eqlist\}\ \}
\end{aligned}$$

We again make use of *nil* to define the behaviour of *MakeClass* in the case where a class does not inherit from another class. The equation for this case is similar to that defined for *MakeInterface* in the previous section so we will omit it here.

We also define projection functions to project out each component of an ACS.

We use *getName* to extract an ACS's name:

$$getName : ClassSpec \rightarrow Name$$

This is defined by the following equation:

$$\begin{aligned}
&getName(MakeClass(MakeInterface(\\
&\quad name, ename, fields, const, meths, ops), hops, eqlist, varis) = name
\end{aligned}$$

Other projection functions for the other components of a class are defined in the obvious way.

3.3 Java Classes to ACSs

In this section we consider how ACSs can be constructed from Java classes with the addition of equations that model the class' behaviour. We will first look at the more trivial process of translating Java classes to a class interface and then the more complex issue of adding semantics to the interface to create the ACS. We will use a set of simple Java classes based on geometric shapes to illustrate the conversion process.

3.3.1 Transforming a Java Class into an Interface

The translation from a Java class to a class interface is relatively simple. At this point we are only interested in syntax and therefore only simple syntax translation is required. In our model, due to the fact we eventually want to produce an FAS in Maude, we try and translate the Java syntax to something that is closer to the Maude syntax while still retaining a Java feel to the layout of the specification. This is done to allow users of either language to be able to easily read the specification. It should be noted that fields and methods are called using a member access notation [Sun05e]. That is given an object *obj* which has a method *meth* we access that method by the following notation *obj.meth(inputs)*. However commonly when documenting a Java class, the syntax of *meth(inputs)* would be used, ignoring the member access part of the notation.

3.3.2 Shape Interface

`Shape` is a the 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 will first give the original Java class code for the `Shape` class.

```
public class Shape {

  public int number;

  public Shape() {
  }

  public int area(){
    return 0;
  }

  public int perimeter(){
    return 0;
  }

  public int return4(){
    return 4;
  }
}
```

At this stage we are only interested in representing the syntax not the semantics of the class. The `Shape` Interface will look like the following:

```
Interface Shape Extends Object{

    Fields{
        Number : Int .
    }

    Constructors{
        Shape : .
    }

    Methods{
        op area : -> Int .
        op perimeter -> Int .
        op return4 : -> Int .
    }

    Operations{
    }

}
```

Note that we say that this interface extends `Object`. This is the default class that any Java class inherits from. At present we can assume for simplicity that the `Object` interface and ACS is empty and serves no purpose. As such, in some examples we choose to omit the **Extends** part of the specification as any specification without an *Extends* part can be assumed to **Extend** `Object`. As can be seen `Shape` is very general and serves more as a base structure for more complex geometric shape classes. In some respects `Shape` would be better defined as an actual *Java Interface* (not to be confused with our algebraic interface) as most of `Shape`'s methods will be overridden, but we will define `Shape` as a class for the sake of showing a consistent example.

3.3.3 Rectangle Interface

Rectangle will inherit from Shape as it too is a type of Shape. However Rectangle is a more complex Shape and is therefore less general. This means that it will override several of Shape's methods (such as area) with Rectangle's own definitions. Here is the Rectangle Java class code:

```
public class Rectangle extends Shape {

  public int side1;
  public int side2;

  public Rectangle(){
  }

  public Rectangle(int i,int j) {
    side1 = i;
    side2 = j;
  }

  public int area(){
    return side1 * side2;
  }

  public int perimeter(){
    return (2 * side1) + (2 * side2);
  }
}
```

Here is Rectangle's algebraic interface specification:

```
Interface Rectangle Extends Shape{

    Fields{
        side1 : Int .
        side2 : Int .
    }

    Constructors{
        Rectangle : .
        Rectangle : Int Int .
    }

    Methods{
        op area : -> Int .
        op perimeter : -> Int .
    }

    Operations{
    }
}
```

As can be seen this class overrides `Shape`'s **`area`** and **`perimeter`** methods. However it does not override the **`return4`** method of `Shape`, hence `Rectangle`'s **`return4`** method's behaviour will be identical to the **`return4`** method of `Shape`. All of `Shape`'s fields are inherited as fields cannot be overridden. We have introduced two new fields **`side1`** and **`side2`**. All of this is especially important when we define semantics for this class in the ACS.

3.3.4 Square Interface

The final class we will look at is `Square`. This will inherit from `Rectangle`. This means that `Square` is not only a more specialised form of `Shape` but also a more specialised form of `Rectangle`. This will mean it will override some of `Rectangle`'s methods. Here is `Square`'s Java class code:

```
public class Square extends Rectangle{

  public Square(int i) {
  }

  public int area(){
    return side1 * side1;
  }

  public int perimeter(){
    return 4 * side1;
  }
  public void setSide(int a){
    side1=a;
  }

}
```

Here is `Square`'s Algebraic Interface Specification:

```
Interface Square Extends Rectangle{

    Fields{
    }

    Constructors{
        Square : Int .
    }

    Methods{
        op area : -> Int .
        op perimeter : -> Int .
        op setSide : Int -> Int .
    }

    Operations{
    }

}
```

Once again any methods we inherit from `Rectangle`, including any methods that we inherit from `Shape` through `Rectangle` will not appear here and are linked to via the *Extends* statement. We also again override the `area` and `perimeter` methods with `Square`'s own definitions. Finally we introduce a new method called `setSide`. This method only belongs to `Square` and is not part of `Shape` or `Rectangle`.

3.3.5 Algebraic Interface Specifications to Algebraic Class Specifications

The next stage is to convert our new interface specifications into ACSs. To do this we need to define the semantics of each class. We do this equationally. This requires the user of this algebraic system to identify what the equations should be that define the behaviour of all the methods and constructors.

We will look at a simple example of this. Let us look at the Java code for `Square`'s `area` method.

```
public int area(){
   return side1 * side1;
}
```

An equational definition for this is as follows:

```
eq (S).area()q = (S).side1 * (S).side1 .
eq (S).area()o = S .
```

There are several important points to note from this example. In the equations we use the member access notation as discussed earlier. This includes both fields and methods. This is because we are now defining how people would actually use the class. To do this we use a `Square` variable `S` which will need to be defined like in the following statement:

```
var S : Square .
```

Another thing to note is that we now introduce the *q* and *o* notation as discussed earlier to show if we are defining a method's state change functionality or its query return value functionality. In the example above we define the value that `area` returns. As `area` does not change the state of `Square` in any way, it just returns a value, we do not need to provide an equation defining state change using the *o* notation. However we have done so here to illustrate both the query and command parts of method definitions. In the above case the command definition simply returns S (the unchanged state of `Square`. In practice and for the rest of this example we will omit the command definitions when a method does not change the state of the class instance. All the equations and operators that would need to be defined to deal with all this new notation are discussed in chapter 4 when we create the FAS . A lot of this can be automatically generated as we will see in section 5.3.

We will now show what the ACSs looks like for each of our interfaces.

3.3.6 Shape Class Specification

The ACS for the Shape interface is as follows:

```
Class Shape Extends Object{

    Hidden{
        op AShape : -> Shape .
    }

    Fields{
        number : Int .
    }

    Constructors{
        Shape : .
    }

    Methods{
        op area : -> Int .
        op perimeter -> Int .
        op return4 : -> Int .
    }

    Operations{
    }

    Variables{
        var S : Shape .
    }

    Equations{
        eq Shape() = AShape .
        eq (S).area()q = 0 .
        eq (S).perimeter()q = 0 .
        eq (S).return4()q = 4 .
    }
}
```

There are several points to note here. First we have introduced a hidden operator *op AShape* $: \rightarrow$ *Shape* . This effectively allows us to define an empty (default) Shape class instance and can be considered to be a type of constant. This can be seen in the Shape constructor which returns the default AShape operator. The second thing to note is that all the methods in our example are only defined by their query return values. This means that none of the methods change the state of Shape (that Shape has no commands).

3.3.7 Rectangle Class Specification

The ACS for the Rectangle interface is as follows:

```
Class Rectangle Extends Shape{

    Hidden{
        op ARectangle : -> Rectangle .
    }

    Fields{
        side1 : Int .
        side2 : Int .
    }

    Constructors{
        Rectangle : .
        Rectangle : Int Int .
    }

    Methods{
        op area : -> Int .
        op perimeter : -> Int .
    }
    Operations{
    }

    Variables{
        vars I J : Int .
        var R : Rectangle .
    }

    Equations{
        eq Rectangle() = ARectangle .
        eq Rectangle(I,J) =
            ((ARectangle).side1:=(I)).side2:=(J) .
        eq (R).area()q = (R).side1 * (R).side2 .
        eq (R).perimeter()q =
            ((R).side1 * 2) + ((R).side2 * 2) .
    }
}
```

In this specification the empty `Rectangle` class instance is `ARectangle`. This is used in both constructors. The first equation defines the constructor with no arguments. The second one takes in two integer values. It creates the empty `ARectangle` class instance and further defines it by assigning the two integer values to the fields `side1` and `side2` thus effectively defining the `Rectangle`'s dimensions. As with `Shape`, the other two methods, `area` and `perimeter`, only return query values, but now they use the standard formulas for defining the perimeter and area for a rectangle to return a value. Although they are not visible here, due to the *extends* keyword, `Shape`'s `number` field and `return4` methods are inherited by the `Rectangle` class.

3.3.8 Square Class Specification

The ACS for the `Square` interface is as follows:

```
Class Square Extends Rectangle{

    Hidden{
        op ASquare : -> Square .
    }

    Fields{
    }

    Constructors{
        Square : Int .
    }

    Methods{
        op area : -> Int .
        op perimeter : -> Int .
        op setSide : Int -> Int .
    }

    Operations{
    }

    Variables{
        var I : Int .
        var S : Square .
    }

    Equations{
        eq Square(I) = (ASquare).side1:=(I) .
        eq (S).area()q = (S).side1 * (S).side1 .
        eq (S).perimeter()q = (S).side1 * 4 .
        eq (S).setSide(I)q = (S).side1 .
        eq (S).setSide(I)o = (S).side1:=(I) .

    }
}
```

In this specification the empty `Square` class instance is `ASquare`. The constructor only assigns a value to one of the `side` fields, `side1` (which is inherited along with other methods and fields from `Rectangle` and, through `Rectangle`, from `Shape`). A `Square`'s dimensions can be defined from one side alone. Therefore there are new definitions for `area` and `perimeter`. We also have a new method called `setSide`. This has both a state change equation and a query value return equation. Informally this method returns the current `side1` value and assigns a new value to `side1`.

We have now examined each of the geometric shape examples and shown the complete conversion process from Java class to Algebraic Class Specification (ACS).

3.4 Algebraic Structure of Interfaces

In this final section we will examine the work of [STR03] on the Algebraic Structure of Interfaces. We will also look at their work on deriving a simple object-oriented *interface definition language.* We will look at how our algebraic model builds on and extends the work in [STR03]. However we will not look at *interface flattening, joining* and *tagging* until Section 4.4 as it relates more closely to work shown in Chapter 4 on creating FASs.

3.4.1 Basic Structure of an Interface

An interface is defined as follows:

$$Interface = Name + Imports + Body.$$

The *Name* must be a unique identifier for the interface. In Java names are made unique by prepending their package name. The *Imports* are a list of interface names that may be required in the body. These interfaces are stored in an *interface repository.* This allows one interface to import the features of an existing interface. In [STR03], an interface name in the import list need not have a corresponding interface in the repository. However in our work we do not allow this. In Java a missing class causes a compile error. We are only interested in modelling classes that successfully compile (i.e. classes that are syntactically correct) and therefore there will be no missing interfaces. Later we will examine how our ACSs relate to the [STR03] *interface definition language* by defining the body part of the Interface which [STR03] largely omit.

3.4.2 Object-Oriented IDLs

[STR03] includes preliminary work on defining a simple interface definition language (IDL) for object-oriented systems. This IDL captures interactions by seperating the components or methods out into *commands* that can only change the state of the internal implementation, and *queries* that can only return values and are strictly prohibited from changing the state of the internal implementation (however note that [STR03] suggest a notation of combining both which we will look at in Section 3.4.3). The body of this IDL is constructed from the following:

- *data type declarations* of the form:

  ```
  sorts ...,s,...
  constants ...,c: -> S, ...
  operations ...,f:s(1)* ... *s(l) -> s, ...
  ```

  for its data sets, constants and operations.

- *program module declarations* which are split into:
  - the state-altering modules

    ```
    commands ...,p:s(1)* ... * s(m), ...
    ```

  - the state-query modules

    ```
    queries ...,q:s(1)* ...* s(n) -> s, ...
    ```

[STR03] then substitute these declarations for interface bodies in their general IDL which forms an interface with seven declaration sections, of the form:

```
interface       I
import          ...,J,...
sorts           ...,s,...
constants       ...,c: -> s, ...
operations      ...,f: s(1)* ... * s(l) -> s, ...
commands        ...,p: s(1)* ... * s(m), ...
queries         ...,q: s(1)* ... * s(n) -> s, ...
endinterface
```

3.4.3 Queries That Can Change The System State

In our model we do not separate methods into queries and commands but instead allow each method to both return a value and change the state of a class instance (however in practice many methods only either return a value *or* change a class instance). This is similar to [STR03]'s function:

$$Q \ : \ World \times OID \times A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow (World \times A_s).$$

That is a query can return a value (A_s) and a new state (*World*), where OID is an object identifier and World is $World = [OID \rightarrow State]$ which is informally the set of all object states.

3.4.4 Algebraic Class Specification Body

Just as with [STR03]'s IDL our ACSs have both a name and an import list (note Java only allows single inheritance so our model will only have either none or one import name in the import list). However there are several major differences to [STR03]'s model of object-oriented IDLs.

The first relates to *constants*. In our model constants are not assigned a separate section. Instead constants would be added to the operations section and take the form of operations with no inputs.

There are also special types of constants that are added to the **Hidden** section along with other operations that are used to define the internal structure of a class such as a distinguished constant (operation) to represent an empty stack for example. We also omit to state *sorts* as there is only ever one sort introduced in a class specification which has the same name as the class specification itself. We assume its presence to be implicit.

The other change as stated previously are *commands* and *queries* that in our model are combined into one section called **Methods**. However we do not explicitly state that methods can return a new class instance as this is the case for all methods.

The biggest difference between our model and the [STR03] model is that we define the semantics as well as the syntax whereas [STR03] is only concerned with syntax. This means that we add to our body two sections. These are **Equations** which are used to define the semantics of the methods, and constructors, and **Variables** which are used in the equations.

3.5 Automatically Building ACS Specifications

In order to aid in the testing of new features to our specification we designed a program to aid in the automatic generation of ACSs and FASs for a Java class with extra embedded information. We called this program the *Algebraic Specification Generator* (ASG). The use of the program for this purpose is discussed in more detail in section 5.3. The program can also be used to automate the building of the ACS and FAS specifications for a user. This is especially useful for FASs because as we will show in the next chapter a large amount of booking keeping and extra information needs to be generated to model the structure of a class. In this section we will look at how we extract information about a Java class using Reflection and how we then use that information to output an ACS. We have omitted discussing in any detail how we embed equations into Java source code comments and how they are extracted as this is discussed in greater detail in section 5.3. We will also show examples of the code and the output it produces in the case of generating specification details for fields. The code itself is too long to discuss in full in the body of the thesis, however the complete code can be found on the appendix CD and instructions on how to use it in appendix A. We will leave discussion of how the program generates an FAS till section 4.5.

3.5.1 Overview Of The Implemented Algorithm For ACS Generation

There are three phases to the ASG when building an ACS specification.

1. Extracting information about the Java class to be specified both from it's compiled class information and from its source code file.
2. Manipulating the extracted data into an appropriate form for the ACS by converting it to required formats and generating any extra information required.
3. Outputting the newly generated information as an ACS.

Extracting information about a Java class is done in two stages. First the ASG uses the Java Reflection classes to extract the signature of the class. Java Reflection can be used to (amongst other things) obtain information about the structure of a class. The names of the fields, constructors and methods are discovered using reflection. Once these parts of a class have been identified reflection is further used to discover the input parameters of methods and constructors and the types and return types of fields and methods. As we are only interested in public methods, fields and constructors of a class we discard any that are not declared as being `public`

The second stage in extracting information about a class is to extract information from the classes source code. Therefore someone using the program must provide the source code for the class as well. The information we wish to extract are equations defining the behaviour of the methods and constructors of a class. This information is embedded in Javadoc comments and is surrounded by special tags. The information to be extracted is done by pattern matching on these tags. We discuss the format of these tags and the extra embedded information in greater detail in section 5.3.

Once the information has been extracted the ASG then manipulates the data to reformat it into the required format for an ACS and generate any extra information that might be needed. With an ACS very little extra information is needed and relatively few changes need to be made to the extracted data. One of the main things done in this phase of the program is to verify no variable names have been declared in the embedded comments twice. If a variable has been declared twice and is of the same type then the duplicate declaration is simply deleted. If the name is declared twice for two different types of variables then the program renames one of the variables. It then changes any calls to that variable in the equations it was declared for to the new name. This is done by a series of pattern matching to identify any references to the variable in the equations. When generating an FAS a lot more reformatting and extra information needs to be generated and we will discuss this in more detail in section 4.5.

The final phase is to output all this information as an ACS. This is a relatively simple phase that takes the required information and outputs it with extra syntax that makes up the structure of an ACS. For instance methods are outputted as a list and enclosed in the `Methods{...}` syntax. Below is an example of the source code for a Java file with embedded java doc comments.

```
/**
 * <p>Title: AClass</p>
 * <p>Description: A generic Class example</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 <hidden>op AAClass : -> AClass .</hidden>
 */

public class AClass {

  /**
   * An integer field.
   */
  public int afield;

  /**
   * Creates an empty AClass
   <code>
   eq AClass() = AAClass .
   </code>
   */
  public AClass() {
  }

  /**
   * Returns the value 4.
   * @return int
   *
   <code>
   var A : AClass .
   eq (A).return4()q = 4 .
   </code>
   */
  public int return4() {
    return 4;
  }

}
```

One advantage of the ASG is that a user does not need to implement the bodies of methods and constructors in order to generate an FAS. All that needs to be provided are the embedded equations and variables and dummy stubs can be generated for the method and constructor bodies. This allows specifications to be built before a class is fully implemented.

The corresponding ACS generated by the ASG for the above class is as follows.

```
Class AClass {
    Constructors{
        AClass :  .
    }
    Fields {
        afield : Int .
    }
    Hidden{
        op AAClass : -> AClass .
    }
    Methods{
        return4 :  -> int .
    }
    Operations{
    }
    Variables{
        var A : AClass .
    }
    Equations{
        eq AClass() = AACLass .
        eq (A).return4()q = 4 .
    }
}
```

3.5.2 Field Example

In this section we will look at some examples of ASG source code for extracting information about fields from Java classes and generating ACS information for them. We will not be showing the complete code as it is very long but will instead be focussing on key sections of it. Let us consider the following section of code.

```
    Field m[] = theclass.getFields();

      for (int i = 0; i < m.length; i++) {
        f = m[i];

        f.getName();
        type = f.getType().getName();

        if (type.compareTo("int") == 0) {
          type = "Int";
        }

        if (type.compareTo("boolean") == 0) {
          type = "Bool";
        }
```

In the above code `theClass` is class instance of type `Class` which is the Java Reflection representation of the class we are modelling. The method `getFields` returns an array of `Fields` which are the Reflection representation of the class' fields. Each of these instances of `Field` can be used to extract information about the fields in the class. We use getName to extract the field's name and getType to discover the field's type. The two if statements are used to check to see if the types of the fields are int or boolean. These are not the correct format for these types in Maude so if they are found to be so we need to change the types to Int and Bool respectively. Field information is then stored in an array of FieldDecs which is a custom class used to gather together all the information discovered for each field.

Very little else needs to be done on the fields at this point and all the information gathered can be output as the ACS representation of a field as shown in the code below.

```
out.println("\Fields{");
for (int k = 0; k < fields.size(); k++) {
  FieldDecs field = (FieldDec) fields[k];
  out.println("\t\t" + field.name + ":" field.type);
}
```

This simply prints out each fields name and type in the Fields section of an ACS.

Therefore the line public int afield; in the Java source code becomes the following in the ACS.

```
Fields {
    afield : Int .
}
```

3.6 Sources

The biggest influence on the work in this chapter is that of [Ree01, STR03]. We discussed the relationship between our work and theirs in section 3.4. The actual structure and design of our Algebraic Class Interfaces and Specifications are influenced by syntax of Java itself [Sun05g] although the over all concept is the work of JB. Also as we will see in the next chapter the next stage in our work is to take the Algebraic Class Specification and produce a Full Algebraic Specification written in Maude. Therefore the structure and design of an ACS has been influenced by that of the Maude language [SRI05]. The code written to automatically build an ACS from a Java class is the work of JB.

Chapter 4

From Algebraic Class Specifications to Full Algebraic Specifications

In this chapter we will examine the next stage in our modelling process where we expand an *Algebraic Class Specification* ACS into a *Full Algebraic Specification* FAS written in *Maude.* We saw in chapter 3 how to create an ACS by extracting syntactical information from a Java class and adding semantic equations to define the behaviour of the methods and constructors. As we discussed in chapter 3, the ACS is not a complete algebraic specification and much more information has to be added in order to produce the FAS. This chapter will show how in order to create the FAS we need to provide a great many more operators and equations to fully model the structure of a class. We have chosen Maude to write the full specification in. As a consequence of using Maude we will be able to execute the FAS, which is useful for testing and analysis purposes. As will be seen in this chapter a large amount of book keeping and extra information needs to be generated in order to create the FAS and it is important to be able to automate as much of this as possible. Therefore we will be discussing the automatic generation of FAS information later in this chapter. The main aim of this chapter is to show the process of transforming an ACS into an FAS.

Throughout this chapter we will be describing the process of translating an ACS into an FAS. For each concept in the FAS we will be presenting examples of the concrete syntax for an FAS both in the general and in specific example cases. We do not make use of any abstract syntax in this chapter. As the translation process is long and complex we will only be demonstrating the formal operators and equations for the implementing algorithm needed to translate ACS methods to FAS methods. For all other concepts we will informally describe the translation algorithm for each case. The complete translation algorithm implemented in Java for all concepts discussed in this chapter can be found in Appendix A.

Section 4.1 will show the example ACS that we are going to use to illustrate the principles discussed in this chapter. It also shows the resulting FAS. The aim of this section is to give the reader a complete view of what we will be defining in this chapter. Section 4.2 will identify and explain the extra operators and equations that need to be created to transform a class specification into a FAS (i.e. one without inheritance). We will also look at how we need to change the format of some of the existing equations and operators. The section will focus on the modelling of the methods of a class, taking into account the fact that methods need to be able to return both a query value and change the state of the class instance. We will also show the formal operators and equations needed to convert ACS methods to FAS methods and give formal definitions for parts of the structure of an FAS. Unfortunately, due to time restraints these formal equations will only deal with aspects relating to ACS to FAS method conversion to demonstrate how we convert a component of an ACS to an FAS. We have chosen methods as an example as they represent a non trivial conversion process. In future work we would wish to define formal operators for the entire ACS to FAS conversion process. We will also show in this section how we implement class fields in our FAS by creating special operators for them. Finally the section will look at how we model the class constructors. This section will not deal with inheritance as that will be dealt with separately in the next section. Section 4.3 will look at how we transform an ACS that models a class which inherits from another class, into an FAS. This involves the concept of flattening the class inheritence tree. We will show all the extra equations and operators we need to generate in order to handle the inheritance behaviour of a class. We will also show the order in which classes with inheritance need to be processed in order to generate the FAS for a class that inherits from another class. We will mostly use a class with only one level of inheritance to demonstrate the process in this section. However we will at the end of the section look at an example that has more than one level of inheritance and show how we accommodate this in our model. Section 4.4 will examine the work of [STR03] on interface flattening, joining and tagging and how it relates to our work. We will show that when we model inheritance in our work, we are using a similar concept to interface flattening in their work. In our work, we flatten the inheritance tree for a class, tagging inherited classes members with the keyword *super*. We will also explain how the way [STR03] join interface bodies together relates to the way our model combines the semantics of an inherited class with that of an inheriting class. Finally in section 4.5 we will continue to examine the code we have written for automating the building of specifications. We will look specifically at how it can build an FAS specification based on the information extracted earlier, as discussed in section 3.5.

By the end of this chapter, combined with the work in the previous chapter, we will have shown the whole process of taking a Java class, generating an ACS from it, and then generating an FAS written in Maude.

4.1 Example

In this section we will show the example `Shape` as an ACS and the FAS that will be produced from it. Throughout the remainder of this chapter we will show how we generate the FAS. Consider a simplified version of the `Shape` class specification.

```
Class Shape Extends Object{

    Hidden{
        op AShape : -> Shape .
    }

    Fields{
        number : Int .
        side1 : Int .

    }

    Constructors{
        Shape : .
    }

    Methods{
        op return4 : -> Int .
    }

    Operations{
    }

    Variables{
        var S : Shape .
    }

    Equations{
        eq Shape() = AShape .
        eq (S).return4()q = 4 .
    }
}
```

This example has been simplified to have one constructor, two fields and one method. The corresponding FAS is show below.

```
fmod SHAPE is

    protecting BUILDLINK .

    sort Shape .
    sort ShapeInt .

    op Shape() :  -> Shape .

    op AShape : -> Shape .

    op _,_ : Shape Int -> ShapeInt .
    op oval(_) : ShapeInt -> Shape .
    op qval(_) : ShapeInt -> Int .
    op _.return4() : Shape  -> ShapeInt .
    op _.return4() : ShapeInt  -> ShapeInt .
    op _.return4()o : Shape -> Shape .
    op _.return4()q : Shape -> Int .

    op _.number : Shape -> Int .
    op _.number:=_ : Shape Int -> Shape .
    op _.number:=_ : ShapeInt Int -> Shape .
    op _.number : ShapeInt  -> Int .
    op _.side1 : Shape -> Int .
    op _.side1:=_ : Shape Int -> Shape .
    op _.side1:=_ : ShapeInt Int -> Shape .
    op _.side1 : ShapeInt  -> Int .
    var S : Shape .
    var SYS0 : Shape .
    var SYS1 : Int .
    var SYS2 : ShapeInt .
    var SYS3 : Shape .
    var SYS4 : Shape .
    var SYS6 : Int .
    var SYS7 : Int .
    var SYS8 : Int .
    var SYS9 : Int .
    var SYS10 : Int .
    var SYS11 : Int .

    eq Shape() = AShape .
    eq (S).return4()q = 4 .
    eq oval(SYS0,SYS1) = SYS0 .
    eq qval(SYS0,SYS1) = SYS1 .
    eq (SYS0).return4() = (SYS0).return4()o,(SYS0).return4()q .
    eq (SYS2).return4() =
        (oval(SYS2)).return4()o,(oval(SYS2)).return4()q .

    eq ((SYS0).number:=(SYS8)).number = SYS8 .
    eq (SYS2).number:=(SYS8) = (oval(SYS2)).number:=(SYS8) .
    eq (SYS2).number = (oval(SYS2)).number .
    eq ((SYS0).side1:=(SYS9)).number = (SYS0).number .
    eq ((SYS0).side1:=(SYS10)).side1 = SYS10 .
    eq (SYS2).side1:=(SYS10) = (oval(SYS2)).side1:=(SYS10) .
    eq (SYS2).side1 = (oval(SYS2)).side1 .
    eq ((SYS0).number:=(SYS11)).side1 = (SYS0).side1 .

endfm
```

We will examine each aspect of this example later in detail. As can be seen the FAS has many more equations and operators than the ACS. Many of the methods have been changed in format to more directly match the object-oriented operator format (in fact the above example is a simplified version of the FAS format, with many operators and equations left out. For example, it does not include the operators and equations that define array construction and access).

4.2 Basic Class Specification Conversion

In this section we will examine the basic conversions that need to be done to convert a stand alone ACS to an FAS. We will deal with how we model inheritance in section 4.3, so in this section we are only interested in the stand-alone class. We will look chiefly at how we model methods that can both change the state of a class instance and return a value. We will also look at field access and assignment, and sorts.

4.2.1 In-built functionality

We will first examine the first line in the above FAS example:

```
fmod SHAPE is protecting BUILDLINK
```

The first two words identify that we are creating a new functional Maude module called `Shape`. The rest of the line allows us to access predefined functions and equations contained in other Maude modules. We will look at the inbuilt functionality in more detail in Chapter 5.3. The complete listing for the `BUILDLINK` module can be found on the Appendix CD at the end of this thesis.

4.2.2 Modelling Methods

As we saw in the last chapter, methods can both change the state of a class instance and return a value. Effectively their return type is *State* × *Value*. However Maude cannot directly model functions that return two types. In order to do so we need to create a tuple of the two types that can be returned and create construction and extraction operators to handle this. When it comes to the equations to define each return type we have already created seperate notation for each through the *method(inputs)o* and the *method(inputs)q* notation discussed in the previous chapter. What we have to do at this stage is provide all the extra operators and equations that join it all together. We will also need to change the syntax of each method to more closely reflect the object-oriented member access notation.

We will use the **return4** method from the above ACS example and show its corresponding FAS functions and equations that are generated for it as shown in the FAS example. The ACS defines **return4** by the following two lines:

```
op return4 : -> Int .
eq (S).return4()q = 4 .
```

So for every method belonging to a class, which we will call *AClass* in general, we will need to make the following conversions and additions.

The format of the method operator in the ACS is informally as follows:

$$op\ method\ : InputField^* \rightarrow ReturnType.$$

This will need to be changed to the following format:

$$op\ \_.method(\_, \ldots, \_)\ :\ AClass\ InputField^*\ \rightarrow\ ReturnType\ . \tag{4.1}$$

So op `return4 : -> Int .` becomes the line from the FAS example:

```
op _.return4() : Shape -> Int .
```

The first thing to note is that an input has been added before the '.' notation. This input is a class instance variable of the class itself. This allows us to use the object-oriented member access notation. The original inputs are contained within the round brackets (although the **return4** example does not actually have any extra inputs). The final thing to note is that the return type has changed. This new return type will be the tuple containing both the state change return type and the query return type. These tuples and relevant operations and equation only need to be declared once for each different query return type that a class' methods can return. These tuples are necessary in order to model methods that both return a value and change the state of a class instance (although in practice we use the tuples for all methods). The Maude language can not handle methods that return two types. To get around this problem we define a tuple as a new sort type which combines the two individual method return types together. For instance if we have a method that changes the state of a class called `AClass` and returns an integer value then we define a sort type `AClassInt` which, informally is formed as follows `(aclass,int)`.

In order to model this we will need to add a new sort which will be the tuple sort type.

$$sort\ AClassReturnType\ . \tag{4.2}$$

In the FAS example this generates the line:

```
sort ShapeInt .
```

We then need to add an operator that will allow us to create an instance of this new return type tuple as well operators that will allow us to extract the components of a tuple instance. In particular we will need to extract the class instance part of the tuple in order to be able to pass it into other methods as ultimately the $method_q$ and $method_o$ style method operators are enacted upon the class instances alone, not the tuple types.

$$op \ \_,\_ \ : \ AClass \times ReturnType \ \longrightarrow \ AClassReturnType \ . \tag{4.3}$$

$$op \ oval(\_) \ : \ AClassReturnType \ \longrightarrow \ AClass \ . \tag{4.4}$$

$$op \ qval(\_) \ : \ AClassReturnType \ \longrightarrow \ ReturnType \ . \tag{4.5}$$

In the FAS example these are the lines:

```
op _,_ : Shape Int -> ShapeInt .
op oval(_) : ShapeInt -> Shape .
op qval(_) : ShapeInt -> Int .
```

Next we will need to define the equations that define the behaviour of these new operators.

$$eq \ oval(aclass, retype) \ = \ aclass \ . \tag{4.6}$$

$$eq \ qval(aclass, retype) \ = \ retype \ . \tag{4.7}$$

In the FAS example these are the lines:

```
eq oval(SYS0,SYS1) = SYS0 .
eq qval(SYS0,SYS1) = SYS1 .
```

Next we may have several equations in the ACS defining the state and query return values for each method using the o and q notation. We will need therefore to define actual operators for these.

$$op \ \_.method(\_,\ldots,\_)o \ : \ AClass \times InputField^{*} \longrightarrow \ AClass \ . \tag{4.8}$$

$$op \ \_.method(\_,\ldots,\_)q \ : \ AClass \times InputField^{*} \longrightarrow \ ReturnType \ . \tag{4.9}$$

In the FAS example these are the lines:

```
op _.return4()o : Shape -> Shape .
op _.return4()q : Shape -> Int .
```

Another important equation we need to add is one that links the original method operator with the *method(inputs)o* and *method(inputs)q* operators.

$$\begin{aligned} &eq\ (aclass).method(inputs)\ = \\ &\quad (aclass).method(inputs)o, (aclass).method(inputs)q. \end{aligned} \tag{4.10}$$

In the FAS example this is the line:

```
eq (SYSO).return4() = (SYSO).return4()o,(SYSO).return4()q .
```

As mentioned earlier in Section 3.2.5 we use a special sort type called `void` to model the behaviour of methods that do not return a value (`void` methods). This sort type is defined in the usual way in the in-built functionality contained in the `BUILDLINK` module. It is defined as follows:

```
sort void .
```

A `void` method is defined as returning a tuple consisting of the new class instance and `void`.

```
op _.ameth(_) : AClass Int -> AClassvoid .
```

As with all other methods, a `void` method is defined as consisting of the values of its *o* and *q* method definitions.

```
eq (aclass).ameth(i) = (aclass).ameth(i)o,(aclass).ameth(i)q .
```

Where in particular `ameth(i)q` is declared with the following operator.

```
op _.ameth(i)q : AClass Int -> void .
```

However as a `void` method does not return a value, the behaviour of `ameth(i)q` will be left undefined. Therefore it will never be evaluated and so it will not return an actual value for the `void` part of the tuple.

Next we have to add extra method operators that can take in any other of the class' tuple types including the one we have just created as its first class instance argument in the member accessor notation. The reason for this is that each method can potentially return one of the classes other tuple types as a class instance. This tuple type contains a valid class instance and therefore needs to be passable in some form to other class methods. An alternative way of doing this would be requiring the user to use the `oval` methods to extract the actual class instance part of the tuple. However we have chosen to try and distance the user as much as possible from the internal workings and operations of an FAS. Therefore we have decided to generate extra operators and equations for each of the tuple types to allow them to be passed directly to the class' methods. For example, with *AClass* we would still need to add the following operator for *AClassReturnType*:

$$\begin{aligned} &op\ \_.method(\_,\ldots,\_)\ :\ AClassReturnType\ InputField^{*}\ \rightarrow \\ &\qquad AClassReturnType\ . \end{aligned} \tag{4.11}$$

There is only one method in our example so we only have one extra method operator needed for the tuple we have just created.

```
op _.return4() : ShapeInt  -> ShapeInt .
```

The final aspect of this we need to add is the equation that links this method operator with the *method(inputs)o* and *method(inputs)q* operators.

$$\begin{aligned} &eq\ (aclassretype).method(inputs)\ = \\ &\quad (oval(aclass)).method(inputs)o,\ (qval(aclass)).method(inputs)q. \end{aligned}$$

In the FAS example this is the line:

```
eq (SYS2).return4() =
   (oval(SYS2)).return4()o,(oval(SYS2)).return4()q .
```

Finally the equations from the ACS are copied into the FAS unchanged. If a method only defines either the state change of a class instance or a query return value, the equation defining either the query value or the class instance value needs not to be defined. This is because there will never be a need to evaluate the missing return type as that is not in the functionality of the method.

So in the FAS example we copy the equation:

```
eq (S).return4()q = 4 .
```

The only thing we have omitted from this conversion is the declaration of all the extra variables used in the equations as this is a trivial addition.

4.2.3 Method Conversion Operators

In this section we will show the formal functions needed for the conversions shown in the previous section. As already stated, due to time limitations we have only been able to produce the formal functions for the previous section. We have chosen to present the operators and equations for method conversion as we feel this represents a non-trivial conversion example.

The first part of this section gives the concrete syntax for the operators and equations that we will need in the FAS to model each method. The second part shows the equations for translating an ACS method into and FAS method. These equations are a component part of the overall algorithm for translating an entire ACS into an FAS.

Before we begin there are a few details to note. The first is that we treat the types *Sort* and *Name* as being equivalent which allows us to concatenate *Names* and *Sorts* together (we will discuss this later). The other thing to note is the use of the ⊔ notation. We use this to denote underscores which are copied in as literal characters. In order to provide explanation for the following operators and equations we refer back to formulae discussed in the previous section. We also again make use of the **Bold** typeface to denote literal strings to be copied in verbatim.

First we present all the operators and equations which are used to define elements of the concrete syntax of an FAS relating specifically to methods.

The following operator is used to define an FAS method operation as shown and explained earlier in formula 4.1.

$$\begin{aligned}&FASMethodOp = Name \times Sort \times Sort^* \times Sort\\ &\mathbf{op}\ \sqcup\ \_\_(\sqcup, \ldots, \sqcup)\ :\ \_\times\_\rightarrow\ \_\\ &\quad : Name \times Sort \times Sort^* \times Sort \rightarrow FASMethodOp\end{aligned}$$

The following operator is used to define an FAS tuple sort type as shown and explained earlier in formula 4.2 (in actual fact it can define any FAS sort type although we only use it for the tuple sorts in this section).

$$\begin{aligned}&FASSort = Name\\ &\mathbf{sort}\ \_\ :\ Name \rightarrow FASSort\end{aligned}$$

The following operator is used to define the structure of an FAS tuple type as shown and explained earlier in formula 4.3.

$$\begin{aligned}&FASTupleOp = Sort \times Sort \times Sort\\ &\mathbf{op}\ \sqcup, \sqcup\ :\ \_\times\_\rightarrow\_\ :\ Sort \times Sort \times Sort \rightarrow FASTupleOp\end{aligned}$$

The following operator is used to define the *qval* operator as shown and explained in formula 4.5.

$$FASQvalOp = Sort \times Sort$$
$$\textbf{op qval}(\sqcup) \ : \ \_\!\rightarrow\!\_ \ : \ Sort \times Sort \rightarrow FASQvalOp$$

The following operator is used to define the *oval* operator as shown and explained in formula 4.4.

$$FASOvalOp = Sort \times Sort$$
$$\textbf{op oval}(\sqcup) \ : \ \_\!\rightarrow\!\_ \ : \ Sort \times Sort \rightarrow FASOvalOp$$

The following operator defines the equation for modelling the behaviour of the *qval* operation and is defined in formula 4.7.

$$FASQvalEq = Name \times Name \times Name$$
$$\textbf{eq qval}(\_,\_) \ = \_ \ : \ Name \times Name \times Name \rightarrow FASQvalEq$$

It should be noted that the general definition of an equation (as discussed in section 3.2.9) is informally as follows:

$$eq\ lhs \ = \ rhs$$

Where *lhs* and *rhs* are *terms*. However the definition of terms would be broad and is already well defined (we ourselves make use of the Maude language definition of equations [CDE$^+$04]). For the purposes of this section we will give definitions for specific formats of equations which relate to the modelling of FAS methods. These can be considered to be more specialised forms of the general format for an equation.

The following operator defines the equation for modelling the behaviour of the *oval* operation and is defined in formula 4.6.

$$FASOvalEq = Name \times Name \times Name$$
$$\textbf{eq oval}(\_,\_) \ = \_ \ : \ Name \times Name \times Name \rightarrow FASOvalEq$$

The following operator is used to define the FAS method state change operator as shown and explained in formula 4.8.

$$FASMethodOOp = Name \times Sort \times Sort^{*} \times Sort$$
$$\textbf{op} \ \sqcup .\_(\sqcup, \ldots, \sqcup)\textbf{o} \ : \ \_\!\times\!\_\!\rightarrow\!\_$$
$$: \ Name \times Sort \times Sort^{*} \times Sort \rightarrow FASMethodOOp$$

The following operator is used to define the FAS method query return value operator as shown and explained in formula 4.9.

$$FASMethodQOp = Name \times Sort \times Sort^{*} \times Sort$$
$$\textbf{op} \ \sqcup \,.\_(\sqcup, \ldots, \sqcup)\mathbf{q} \ : \_\times\_\rightarrow\_$$
$$: \ Name \times Sort \times Sort^{*} \times Sort \rightarrow FASMethodQOp$$

The following operator defines the equation for modelling the linking of a method's state change and query return values into a tuple type as shown and explained in formula 4.10.

$$FASDefaultMethLinkEq =$$
$$Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*}$$
$$\textbf{eq} \ (\_).\_(\_) = (\_).\_(\_)\mathbf{o}, (\_).\_(\_)\mathbf{q} \ :$$
$$Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*}$$
$$\rightarrow FASDefaultMethLinkEq$$

The following operator defines the equation for modelling the linking of a methods state change and query return values into a tuple type, which takes in a tuple type as the class instance as shown and explained in formula 4.11.

$$FASTupleMethLinkEq =$$
$$Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*}$$
$$\textbf{eq} \ (\_).\_(\_) = (\mathbf{oval}(\_)).\_(\_)\mathbf{o}, (\mathbf{qval}(\_)).\_(\_)\mathbf{q} \ :$$
$$Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*} \times Sort \times Name \times Sort^{*}$$
$$\rightarrow FASTupleMethLinkEq$$

The actual ACS method equations are copied in from the ACS to the FAS unchanged and hence we do not need to define the format for it here.

Now we have defined the concrete syntax for all the operators and equations needed to define a method in an FAS we will now show how we generate these by translating an ACS method into an FAS method. This translation is done through a series of operators and equations which we will define below. These form part of the complete translation algorithm for transforming an ACS into an FAS. The complete code for this is given in Appendix A. The operator and equations for transforming an ACS method to an FAS method as shown and explained in formula 4.1 is as follows.

$$ACSMethodToFAS : Sort \times Method \longrightarrow FASMethodOp$$
$$ACSMethodToFAS(aclass, \textbf{op}\ name : inputfields \longrightarrow retfield) =$$
$$\textbf{op\_}.name(Spacer(inputfields)) \ : aclass \times inputfields \longrightarrow aclass + retfield$$

Where $aclass \in Sort$, $name \in Name$, $inputfields \in Sort^*$, $retfield \in Sort$.

The function *Spacer* is defined as follows:

$$Spacer : Sort^* \longrightarrow String$$
$$Spacer(emptyfield) = ''$$
$$Spacer(afield) = '\sqcup '$$
$$Spacer(afield \times fieldlist) = '\sqcup \, , ' + Spacer(fieldlist)$$

Where $emptyfield \in Sort^0$ and denotes an empty *Sort* set, $afield \in Sort$, and $fieldlist \in Field^+$ (i.e. a set of one or more *Fields*, see chapter 3 for discussion on the * and $^+$ notation). Spacer is used to define the series of underscores to represent an FAS method operator's input arguments.

The operator and equation to generate the FAS tuple sort type as shown in formula 4.2 is as follows.

$$GenFASTuple : Sort \times Sort \longrightarrow FASSort$$
$$GenFASTuple(aclass, afield) = \textbf{sort}\ aclass + afield$$

Where $aclass \in Sort$ and $afield \in Sort$. The above equation uses the operator $+$ on *aclass* and *afield* to join their literal string names together. A *Sort* or a *Name* can be considered to consist of simply a literal string identifier. For instance the operators to generate *Sorts* and *Names* are informally as follows.

$$op\ \_ \ : \ String \longrightarrow Sort$$

$$op\ \_ \ : \ String \longrightarrow Name$$

The $+$ operator, which essentially is used to represent string concatenation, can have as inputs both *Sorts* and *Names*. We assume $+$ is able to project out the string component of *Sorts* and *Names* when passed as arguments (remember we treat *Sorts* as being equivalent to *Names*). For example given two input sorts called *AClass* and *Int* then $+$ will produce the result *AClassInt*.

In order to generate tuple types we need to know the return type of a method. We use the following operation to discover this.

$$\begin{aligned}&GetSingleRetType : Method \rightarrow Sort\\&GetSingleRetType(\mathbf{op}\ name : inputfields{\rightarrow}rettype) = rettype\end{aligned}$$

Where $rettype \in Sort$. It should be noted that this and other equations could be simplified with the use of projection functions on ACS classes and their components.

We need to examine all of an ACS's methods and create a list of sorts representing all their return types. We use the following operation to do this.

$$\begin{aligned}&GetMethRetTypes : Sort^* \times Method^* \rightarrow Sort^*\\&GetMethRetTypes(sortlist, EmptyMethods) = sortlist\\&GetMethRetTypes(sortlist, methodlist) =\\&\quad GetMethRetTypes(GetSingleRetType(head(methodlist))+\\&\quad sortlist, tail(methodlist))\\&\quad (if GetSingleRetType(head(methodlist))\ not \in sortlist)\\&GetMethRetTypes(sortlist, methodlist) =\\&\quad GetMethRetTypes(sortlist, tail(methodlist))\\&\quad (if GetSingleRetType(head(methodlist)) \in sortlist)\end{aligned}$$

Where *head* and *tail* can be assumed to be standard list operators, $sortlist \in Sort^*$, $methodlist \in Method^*$, and $EmptyMethods \in Method^0$. In this case $+$ is used to as a list addition operator by adding a new element to a list. It should be noted that when we generate the tuple sorts and operations we only want to create one for each return type. Therefore if for example two methods return an *int* then we only want to add one reference to *int* to the sort list returned by *GetMethRetTypes*. Therefore the above operation checks to see if a sort type is already in the sort list to be a returned and only adds the sort if it doesn't already exist in the list.

The following operation takes in a list of methods and returns a list of *FASSorts* as shown and explained in formula 4.2.

$$\begin{aligned}&GetAllTupleSorts : Sort \times Method^* \rightarrow FASSort^*\\&GetAllTupleSorts(aclass, methodlist) =\\&\quad GetAllTupleSortsAux(aclass, GetMethRetTypes(methodlist))\end{aligned}$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GetAllTupleSortsAux* which is defined as follows:

$$\begin{aligned}
&GetAllTupleSortsAux : Sort \times Sort^* \rightarrow FASSort^*\\
&GetAllTupleSortsAux(aclass, EmptySorts) = EmptyFasTupleSorts\\
&GetAllTupleSortsAux(aclass, sortlist) =\\
&\quad GenFASTuple(aclass, head(sortlist)) +\\
&\quad GetAllTupleSortsAux(aclass, tail(sortlist))
\end{aligned}$$

Where $EmptySorts \in Sort^0$ and $EmptyFASTupleSorts \in FASSort^*$. This takes a list of *Sorts* and returns the *FASSorts*.

The following operation takes in a list of methods and returns a list of *FASTupleOps* as shown and explained in formula 4.3.

$$\begin{aligned}
&GetAllTupleOps : Sort \times Method^* \rightarrow FASTupleOp^*\\
&GetAllTupleOps(aclass, methodlist) =\\
&\quad GetAllTupleOpsAux(aclass, GetMethRetTypes(methodlist))
\end{aligned}$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GetAllTupleOpsAux* which is defined as follows:

$$\begin{aligned}
&GetAllTupleOpsAux : Sort \times Sort^* \rightarrow FASTupleOp^*\\
&GetAllTupleOpsAux(aclass, EmptySorts) = EmptyFasTupleOps\\
&GetAllTupleOpsAux(aclass, sortlist) =\\
&\quad GenTupleOp(aclass, head(sortlist)) +\\
&\quad GetAllTupleOpsAux(aclass, tail(sortlist))
\end{aligned}$$

Where $EmptySorts \in Sort^0$ and $EmptyFASTupleOps \in FASTupleOp^*$. This takes a list of *Sorts* and returns the *FASTupleOps*.

The following operation takes in a list of methods and returns a list of *FASOvalOps* as shown and explained in formula 4.4.

$$\begin{aligned}
&GetAllOvalOps : Sort \times Method^* \rightarrow FASOvalOp^*\\
&GetAllOvalOps(aclass, methodlist) =\\
&\quad GetAllOvalOpsAux(aclass, GetMethRetTypes(methodlist))
\end{aligned}$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GetAllOvalOpsAux* which is defined as follows:

$$
\begin{aligned}
&GetAllOvalOpsAux : Sort \times Sort^{*} \rightarrow FASOvalOp^{*}\\
&GetAllOvalOpsAux(aclass, EmptySorts) = EmptyFasOvalOps\\
&GetAllOvalOpsAux(aclass, sortlist) =\\
&\quad GenOvalOp(aclass, head(sortlist)) +\\
&\quad GetAllOvalOpsAux(aclass, tail(sortlist))
\end{aligned}
$$

Where $EmptySorts \in Sort^{0}$ and $EmptyFASOvalOps \in FASOvalOp^{*}$. This takes a list of *Sorts* and returns the *FASOvalOps*.

The following operation takes in a list of methods and returns a list of *FASQvalOps* as shown and explained in formula 4.5.

$$
\begin{aligned}
&GetAllQvalOps : Sort \times Method^{*} \rightarrow FASQvalOp^{*}\\
&GetAllQvalOps(aclass, methodlist) =\\
&\quad GetAllQvalOpsAux(aclass, GetMethRetTypes(methodlist))
\end{aligned}
$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GetAllQvalOpsAux* which is defined as follows:

$$
\begin{aligned}
&GetAllQvalOpsAux : Sort \times Sort^{*} \rightarrow FASQvalOp^{*}\\
&GetAllQvalOpsAux(aclass, EmptySorts) = EmptyFasQvalOps\\
&GetAllQvalOpsAux(aclass, sortlist) =\\
&\quad GenQvalOp(aclass, head(sortlist)) +\\
&\quad GetAllQvalOpsAux(aclass, tail(sortlist))
\end{aligned}
$$

Where $EmptySorts \in Sort^{0}$ and $EmptyFASQvalOps \in FASQvalOp^{*}$. This takes a list of *Sorts* and returns the *FASQvalOps*.

The following operation generates the *FASTupleOp* as defined and explained in formula 4.3.

$$
\begin{aligned}
&GenTupleOp : Sort \times Sort \rightarrow FASTupleOp\\
&GenTupleOp(aclass, afield) = \mathbf{op}\ \_,\_\ : aclass \times afield \rightarrow aclass + afield
\end{aligned}
$$

The following operation generates the *FASOvalOp* as defined and explained in formula 4.4.

$$
\begin{aligned}
&GenOvalOp : Sort \times Sort \rightarrow FASOvalOp\\
&GenOvalOp(aclass, afield) = \mathbf{op}\ \mathbf{oval}(\_)\ : aclass + afield\ \rightarrow\ aclass\ .
\end{aligned}
$$

The following operation generates the *FASQvalOp* as defined and explained in formula 4.5.

$$GenQvalOp : Sort \times Sort \longrightarrow FASQvalOp$$
$$GenQvalOp(aclass, afield) = \textbf{op qval(\_)} \ : aclass + afield \ \longrightarrow \ afield \ .$$

The following operation takes in a list of methods and returns a list of *FASOvalEqs* as shown and explained in formula 4.6.

$$GetAllOvalEqs : Sort \times Method^{*} \longrightarrow FASOvalEq^{*}$$
$$GetAllOvalEps(aclass, methodlist) =$$
$$GetAllOvalEqsAux(aclass, GetMethRetTypes(methodlist))$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GetAllOvalEqsAux* which is defined as follows:

$$GetAllOvalEqsAux : Sort \times Sort^{*} \longrightarrow FASOvalEq^{*}$$
$$GetAllOvalEqsAux(aclass, EmptySorts) = EmptyFasOvalEqs$$
$$GetAllOvalEqsAux(aclass, sortlist) =$$
$$GetOvalEqs(aclass, head(sortlist))+$$
$$GetAllOvalEqsAux(aclass, tail(sortlist))$$

Where $EmptySorts \in Sort^{0}$ and $EmptyFASOvalEqs \in FASOvalEq^{*}$. This takes a list of *Sorts* and returns the *FASOvalEqs*.

The following operation takes in a list of methods and returns a list of *FASQvalEqs* as shown and explained in formula 4.7.

$$GetAllQvalEqs : Sort \times Method^{*} \longrightarrow FASQvalEq^{*}$$
$$GetAllQvalEps(aclass, methodlist) =$$
$$GetAllQvalEqsAux(aclass, GetMethRetTypes(methodlist))$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GetAllQvalEqsAux* which is defined as follows:

$$GetAllQvalEqsAux : Sort \times Sort^{*} \longrightarrow FASQvalEq^{*}$$
$$GetAllQvalEqsAux(aclass, EmptySorts) = EmptyFasQvalEqs$$
$$GetAllQvalEqsAux(aclass, sortlist) =$$
$$GetQvalEqs(aclass, head(sortlist))+$$
$$GetAllQvalEqsAux(aclass, tail(sortlist))$$

Where $EmptySorts \in Sort^0$ and $EmptyFASQvalEqs \in FASQvalEq^*$. This takes a list of *Sorts* and returns the *FASQvalEqs*.

The following operation generate the *FASOvalEqs* as shown and described in formula 4.6.

$$\begin{aligned}&GenOvalEq : Sort \times Sort \longrightarrow FASOvalEq\\&GenOvalEq(aclass, rettype) =\\&\quad GenOvalEqAux(GenVarName(aclass), GenVarName(rettype))\end{aligned}$$

This operator and others in this section make use of *GenVarName*. We will not define it here, however informally this operation takes in a sort type and returns the name of a variable of that sort type. It is assumed that this variable name is either added too or obtained from a list of existing variable names.

The above operation takes in two sorts, generates variable names for them and passes them to *GenAllOvalEqAux* which creates an *FASOvalEq* and is defined as follows:

$$\begin{aligned}&GenOvalEqAux : Name \times Name \longrightarrow FASOvalEq\\&GenOvalEqAux(aclassvar, afieldvar) =\\&\quad \textbf{eq oval}(aclassvar,afieldvar) = aclassvar\end{aligned}$$

The following operation generate the *FASQvalEqs* as shown and described in formula 4.7.

$$\begin{aligned}&GenQvalEq : Sort \times Method \longrightarrow FASOvalEq\\&GenQvalEq(aclass, rettype) =\\&\quad GenQvalEqAux(GenVarName(aclass), GenVarName(rettype))\end{aligned}$$

The above operation takes in two sorts, generates variable names for them and passes them to *GenAllQvalEqAux* which creates an *FASQvalEq* and is defined as follows:

$$\begin{aligned}&GenQvalEqAux : Name \times Name \longrightarrow FASQvalEq\\&GenQvalEqAux(aclassvar, afieldvar) =\\&\quad \textbf{eq qval}(aclassvar,afieldvar) = afieldvar\end{aligned}$$

The following operation generates the FAS method's state change operator as shown and explained in formula 4.8:

$$
\begin{aligned}
&GenMethOOp : Sort \times Method \rightarrow FASMethodOOp\\
&GenMethOOp(aclass, \textbf{op}\ name{:}inputfields{\rightarrow}rettype) =\\
&\quad \textbf{op}\ \_.name(Spacer(inputfields))\textbf{o}\ \ : aclass \times inputfields \rightarrow aclass\ .
\end{aligned}
$$

The following operation generates the FAS method's query return operator as shown and explained in formula 4.9:

$$
\begin{aligned}
&GenMethQOp : Sort \times Method \rightarrow FASMethodQOp\\
&GenMethQOp(aclass, \textbf{op}\ name{:}inputfields{\rightarrow}rettype) =\\
&\quad \textbf{op}\_.name(Spacer(inputfields))\textbf{q}\ \ : aclass \times inputfields \rightarrow rettype\ .
\end{aligned}
$$

The following operation generates the *FASDefaultMethLinkEq* as shown and discussed in 4.10.

$$
\begin{aligned}
&GenMethLinkEq : Sort \times Method \rightarrow FASDefaultMethLinkEq\\
&GenMethLinkEq(aclass, \textbf{op}\ name{:}inputfields{\rightarrow}rettype) =\\
&\quad GenMethLinkEqAux(name, GenVarName(aclass),\\
&\quad GenVarNames(inputfields))
\end{aligned}
$$

The above operation takes in a sort and a method, generates variable names for them (we assume *GenVarName* is capable of producing a list of variable names from a list of sorts for the sake of convenience) and passes them to *GenMethLinkEqAux* which creates an *FASDefaultMethLinkEq* and is defined as follows:

$$
\begin{aligned}
&GenMethLinkEqAux : Name \times Name \times Name^*\\
&\quad \rightarrow FASDefaultMethLinkEq\\
&GenMethLinkEqAux(name, aclassvar, inputfieldvars) =\\
&\quad \textbf{eq}\ (aclassvar).name(inputfieldvars)\textbf{o},\\
&\quad (aclassvar).name(inputfieldsvars)\textbf{q}
\end{aligned}
$$

The following operation generates the *FASTupMethLinkEq* as shown and discussed in 4.11.

$$
\begin{aligned}
&GenTupMethLinkEq : Sort \times Method \rightarrow FASTupleMethLinkEq\\
&GenTupMethLinkEq(aclass, \textbf{op}\ name{:}inputfields{\rightarrow}rettype) =\\
&\quad GenTupMethLinkEqAux(name,\\
&\quad GenVarName(aclass), GenVarNames(inputfields))
\end{aligned}
$$

The above operation takes in a sort and a method, generates variable names for them and passes them to *GenTupLinkEqAux* which creates an *FASTupleMethLinkEq* and is defined as follows:

$$
\begin{aligned}
&GenTupMethLinkEqAux : Name \times Name \times Name^* \\
&\quad \rightarrow FASTupleMethLinkEq \\
&GenTupMethLinkEqAux(name, aclassvar, inputfieldvars) = \\
&\quad \textbf{eq}\ (\textbf{oval}(aclassvar)).name(inputfieldvars)\textbf{o}, \\
&\quad (\textbf{oval}(aclassvar)).name(inputfieldsvars)\textbf{q}
\end{aligned}
$$

The following operation takes in a list of methods and defines *FASTupleMethLinkEqs* for each possible tuple type as shown and described in formula 4.11.

$$
\begin{aligned}
&GenAllTupMethLinkEqs : Sort \times Method^* \times Method \\
&\quad \rightarrow FASTupleMethLinkEq^* \\
&GenAllTupMethLinkEqs(aclass, methodlist, amethod) = \\
&\quad GenAllTupMethLinkEqsAux(aclass, \\
&\quad GetMethRetTypes(methodlist), amethod)
\end{aligned}
$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GenAllTupMethLinkEqsAux* which is defined as follows:

$$
\begin{aligned}
&GenAllTupMethLinkEqsAux : Sort \times Sort^* \times Method \\
&\quad \rightarrow FASTupleMethLinkEq^* \\
&GenAllTupMethLinkEqsAux(aclass, EmptySorts, amethod) = \\
&\quad EmptyFASTupleMethLinkEqs \\
&GenAllTupMethLinkEqsAux(aclass, sortlist, amethod) = \\
&\quad GenTupMethLinkEq(aclass + head(sortlist), amethod) + \\
&\quad GenAllTupMethLinkEqsAux(aclass, tail(sortlist), amethod)
\end{aligned}
$$

Where $EmptySorts \in Sort^0$ and
$EmptyFASTupleMethLinkEqs \in FASTupleMethLinkEq^*$. This takes a list of *Sorts* and returns the *FASTupleMethLinkEqs*.

The following operation generates FAS method operators for each of the possible tuple types.

$$\begin{aligned}
&GenAllACStoFASTupMeths : Sort \times Method^{*} \times Method\\
&\quad \rightarrow FASMethodOp^{*}\\
&GenAllACStoFASTupMeths(aclass, methodlist, amethod) =\\
&\quad GenAllACSToFASTupMethsAux(aclass,\\
&\quad GetMethRetTypes(methodlist), amethod)
\end{aligned}$$

The above operation extracts a list of sorts from a list of methods' return types and passes them to *GenAllACStoFASTupMethsAux* which is defined as follows:

$$\begin{aligned}
&GenAllACSToFASTupMethsAux : Sort \times Sort^{*} \times Method\\
&\quad \rightarrow FASMethodOp^{*}\\
&GenAllACSToFASTupMethsAux(aclass, EmptySorts, amethod) =\\
&\quad EmptyFASMethOps\\
&GenAllACSToFasTupMethsAux(aclass, sortlist, amethod) =\\
&\quad ACSMethodToFAS(aclass + head(sortlist), amethod) +\\
&\quad GenAllACSToFASTupMethsEqsAux(aclass, tail(sortlist), amethod)
\end{aligned}$$

Where $EmptySorts \in Sort^{0}$ and $EmptyFASMethodOps \in FASMethodOps^{*}$. This takes a list of *Sorts* and returns the *FASMethodOps*.

4.2.4 Field Operators and Equations

In order to model fields in the FAS we make use of two operations which will be used to define the behaviour of fields. We define equations that allow us to *get* and *set* the field values of a given class instance. 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 as they normally would in Java.

As in the previous section we will demonstrate how we do this, first in the general case using a general class *AClass* and then show how we do it using the `Shape` example above. From the ACS example we define a field:

```
number : Int .
```

First we must define the get and set operations for a field.

$$op\ \_.afield\ :\ AClass\ \rightarrow\ FieldType\ .$$

$$op\ \_.afield := \_\ :\ AClass\ FieldType\ \rightarrow\ AClass\ .$$

In the FAS example this generates the lines:

```
op _.number : Shape -> Int .
op _.number:=_ : Shape Int -> Shape .
```

We also need to create get and set operations that take in each of the tuple types that have been generated for the methods earlier as each of these are potential input types of a class instance using the member access notation:

$$op\ \_.afield\ :\ AClassReturnType\ \rightarrow\ FieldType\ .$$

$$op\ \_.afield := \_\ :\ AClassReturnType\ Int\ \rightarrow\ AClass\ .$$

In the FAS example these are the lines:

```
op _.number:=_ : ShapeInt Int -> Shape .
op _.number : ShapeInt -> Int .
```

Next we create an equation that defines the get and set methods for the operators.

$$eq\ ((aclass).afield := (avalue)).afield\ =\ avalue\ .$$

In the FAS example this is the line:

```
eq ((SYS0).number:=(SYS8)).number = SYS8 .
```

Next we define a couple of equations that define how to deal with any of the tuple types that are passed as class instances to the get and set operators. These use the *oval* operators to unpack the class instance part of the tuples.

$$eq\ (aclassrettype).afield := (avalue)\ =\ (oval(aclassrettype)).afield := (avalue)\ .$$

$$eq\ (aclassrettype).afield\ =\ (oval(aclassrettype)).afield\ .$$

In the FAS example these are the lines:

```
eq (SYS2).number:=(SYS8) = (oval(SYS2)).number:=(SYS8) .
eq (SYS2).number = (oval(SYS2)).number .
```

Finally for every field other than the one we are currently defining, we have to have an equation that allows us to scan through the class instance's term list and ignore the other fields terms.

$$eq\ ((aclass).afield := (fieldtype)).anotherfield \ = \ (aclass).afield\ .$$

This allows us to search for a particular field term within the class instance's term list and discard field terms that are not for the field we are looking for. We need to do this for every alternative field in the class and also to do the same thing for the other fields to. In our FAS example this produces the following equation:

```
eq ((SYS0).number:=(SYS11)).side1 = (SYS0).side1 .
```

There is only one other field, called `side1`, so we only need one equation to deal with this field.

4.2.5 Constructors

Constructors are similar to methods except they only return the class type as there is no query part to constructors. Therefore they do not need to make use of tuples. Also they do not need to take in an instance of the class due to their construction nature.

All constructors will return a new instance of the class initialised to some default setting specified by the constructor's defining equations. Usually this is some operator which symbolises the empty class or default class. This empty class is specified by the user in the **Hidden** section of the ACS.

As in the previous section we will demonstrate how we convert constructors in the ACS to constructors in the FAS. We will first demonstrate a generic example for a class called *AClass*. We will then show how we do the translation for the example `Shape`. In the ACS example there is only one constructor, which is defined by the following lines:

```
Shape : .
eq Shape() = AShape .
```

These in turn are making use of the hidden declaration. The hidden declaration is ported to the FAS without any alteration or additional equations.

```
op AShape : -> Shape .
```

The only changes we make to constructor declarations is to state that it is an *op* and to specifically define the parenthesis structure of the input parameter for the operator declaration. We also define the return type as that of the class to which the constructor belongs.

$$op\ AClass(\_,\ldots,\_)\ :\ InputTypes \rightarrow AClass\ .$$

All other equations are ported to the FAS unchanged. In our example the operator declaration changes to the following:

```
op Shape() :  -> Shape .
```

There are no input types in our example so empty parenthesis are used.

4.3 Inheritance and Class Specification Conversion

In this section we will look at how we model inheritance. The importing of methods from an inherited class is reasonably trivial but the modelling of method overriding and accessing overridden methods is more complex. In Java 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.

To illustrate this we will first use the more complex version of the `Shape` ACS example that we looked at in Section 3.3.6.

```
Class Shape Extends Object{

    Hidden{
        op AShape : -> Shape .
    }

    Fields{
        number : Int .
    }

    Constructors{
        Shape : .
    }
    Methods{
        op area : -> Int .
        op perimeter -> Int .
        op return4 : -> Int .
    }

    Operations{
    }

    Variables{
        var S : Shape .
    }

    Equations{
        eq Shape() = AShape .
        eq (S).area()q = 0 .
        eq (S).perimeter()q = 0 .
        eq (S).return4()q = 4 .
    }
}
```

This using the ACS to FAS conversion methods that we looked at in Section 4.2 will produce the following FAS:

```
fmod SHAPE is

    protecting BUILDLINK .

    sort Shape .
    sort ShapeInt .

    op Shape() :  -> Shape .

    op AShape : -> Shape .

    op _,_ : Shape Int -> ShapeInt .
    op oval(_) : ShapeInt -> Shape .
    op qval(_) : ShapeInt -> Int .
    op _.area() : Shape  -> ShapeInt .
    op _.area() : ShapeInt  -> ShapeInt .
op _.area()o : Shape -> Shape .
op _.area()q : Shape -> Int .
op _.perimeter() : Shape  -> ShapeInt .
op _.perimeter() : ShapeInt  -> ShapeInt .
op _.perimeter()o : Shape -> Shape .
op _.perimeter()q : Shape -> Int .
op _.return4() : Shape  -> ShapeInt .
op _.return4() : ShapeInt  -> ShapeInt .
op _.return4()o : Shape -> Shape .
op _.return4()q : Shape -> Int .

op _.number : Shape -> Int .
op _.number:=_ : Shape Int -> Shape .
op _.number:=_ : ShapeInt Int -> Shape .
op _.number : ShapeInt  -> Int .

op null : -> Shape .

var S : Shape .
var SYS0 : Shape .
var SYS1 : Int .
var SYS2 : ShapeInt .
var SYS3 : Shape .
var SYS4 : Shape .
var SYS5 : ShapeArray .
var SYS6 : Int .
var SYS7 : Int .
var SYS8 : Int .

eq Shape() = AShape .
eq (S).area()q = 0 .
eq (S).perimeter()q = 0 .
eq (S).return4()q = 4 .
eq oval(SYS0,SYS1) = SYS0 .
eq qval(SYS0,SYS1) = SYS1 .
eq (SYS0).area() = (SYS0).area()o,(SYS0).area()q .
eq (SYS2).area() =
    (oval(SYS2)).area()o,(oval(SYS2)).area()q .
eq (SYS0).perimeter() =
    (SYS0).perimeter()o,(SYS0).perimeter()q .
eq (SYS2).perimeter() =
        (oval(SYS2)).perimeter()o,(oval(SYS2)).perimeter()q .
    eq (SYS0).return4() =
        (SYS0).return4()o,(SYS0).return4()q .
    eq (SYS2).return4() =
        (oval(SYS2)).return4()o,(oval(SYS2)).return4()q .

    eq ((SYS0).number:=(SYS8)).number = SYS8 .
    eq (SYS2).number:=(SYS8) = (oval(SYS2)).number:=(SYS8) .
    eq (SYS2).number = (oval(SYS2)).number .

endfm
```

We then extend the Shape ACS with the Rectangle ACS seen in Section 3.3.7.

```
Class Rectangle Extends Shape{

    Hidden{
        op ARectangle : -> Rectangle .
    }

    Fields{
        side1 : Int .
        side2 : Int .
    }

    Constructors{
        Rectangle : .
        Rectangle : Int Int .
    }

    Methods{
        op area : -> Int .
        op perimeter : -> Int .
    }

    Operations{
    }
    Variables{
        vars I J : Int .
        var R : Rectangle .
    }

    Equations{
        eq Rectangle() = ARectangle .
        eq Rectangle(I,J) =
            ((ARectangle).side1:=(I)).side2:=(J) .
        eq (R).area()q = (R).side1 * (R).side2 .
        eq (R).perimeter()q =
            ((R).side1 * 2) + ((R).side2 * 2) .
    }
}
```

The corresponding FAS for Rectangle in which the inheritance is modelled is as follows:

```
fmod RECTANGLE is

    protecting BUILDLINK .

    sort Shape .
    sort Rectangle .
    sort RectangleInt .
    sort ShapeInt .

    subsort RectangleInt < ShapeInt .
    subsort Rectangle < Shape .

    op Rectangle() :  -> Rectangle .
    op Rectangle(_,_) : Int Int -> Rectangle .

    op AShape : -> Shape .
    op ARectangle : -> Rectangle .

    op _,_ : Rectangle Int -> RectangleInt .
    op oval(_) : RectangleInt -> Rectangle .
    op qval(_) : RectangleInt -> Int .
op _,_ : Shape Int -> ShapeInt .
op oval(_) : ShapeInt -> Shape .
op qval(_) : ShapeInt -> Int .

op _.area() : Rectangle  -> RectangleInt .
op _.area() : RectangleInt  -> RectangleInt .
op _.area()o : Rectangle -> Rectangle .
op _.area()q : Rectangle -> Int .

op _.perimeter() : Rectangle  -> RectangleInt .
op _.perimeter() : RectangleInt  -> RectangleInt .
op _.perimeter()o : Rectangle -> Rectangle .
op _.perimeter()q : Rectangle -> Int .

op _.super.area() : Shape  -> ShapeInt .
op _.super.area() : ShapeInt  -> ShapeInt .
op _.super.area()o : Shape -> Shape .
op _.super.area()q : Shape -> Int .

op _.super.perimeter() : Shape  -> ShapeInt .
op _.super.perimeter() : ShapeInt  -> ShapeInt .
op _.super.perimeter()o : Shape -> Shape .
op _.super.perimeter()q : Shape -> Int .

op _.return4() : Rectangle  -> RectangleInt .
op _.return4() : RectangleInt  -> RectangleInt .
op _.return4()o : Rectangle -> Rectangle .
op _.return4()q : Rectangle -> Int .

op _.super.return4() : Shape  -> ShapeInt .
op _.super.return4() : ShapeInt  -> ShapeInt .
op _.super.return4()o : Shape -> Shape .
op _.super.return4()q : Shape -> Int .

op _.side1 : Rectangle -> Int .
op _.side1:=_ : Rectangle Int -> Rectangle .
op _.side1:=_ : RectangleInt Int -> Rectangle .
op _.side1 : RectangleInt  -> Int .

op _.side2 : Rectangle -> Int .
op _.side2:=_ : Rectangle Int -> Rectangle .
op _.side2:=_ : RectangleInt Int -> Rectangle .
op _.side2 : RectangleInt  -> Int .

op _.number : Rectangle -> Int .
op _.number:=_ : Rectangle Int -> Rectangle .
op _.number:=_ : RectangleInt Int -> Rectangle .
op _.number : RectangleInt  -> Int .

var S : Shape .
var SYS0 : Rectangle .
var SYS1 : Shape .
vars I J : Int .
var R : Rectangle .
var SYS2 : Shape .
var SYS3 : Rectangle .
var SYS4 : Int .
var SYS5 : RectangleInt .
var SYS6 : Int .
var SYS7 : ShapeInt .
var SYS8 : Rectangle .
var SYS9 : Rectangle .
var SYS10 : RectangleArray .
var SYS11 : Int .
var SYS12 : Int .
var SYS13 : Int .
var SYS14 : Int .
var SYS15 : Int .
var SYS16 : Int .
var SYS17 : Int .
var SYS18 : Int .
var SYS19 : Int .
var SYS20 : Int .
var SYS21 : Int .

eq (S).super.area()q = 0 .
eq (S).super.perimeter()q = 0 .
eq (S).super.return4()q = 4 .

eq (R).area()q = 0 [owise] .
eq (R).perimeter()q = 0 [owise] .
eq (R).return4()q = 4 [owise] .

eq Rectangle() = ARectangle .
eq Rectangle(I,J) = ((ARectangle).side1:=(I)).side2:=(J) .

eq (R).area()q = (R).side1 * (R).side2 .
eq (R).perimeter()q = ((R).side1 * 2) + ((R).side2 * 2) .

eq oval(SYS3,SYS4) = SYS3 .
eq qval(SYS3,SYS4) = SYS4 .
eq oval(SYS2,SYS6) = SYS2 .
eq qval(SYS2,SYS6) = SYS6 .

eq (SYS1).super.return4()q = (SYS1).return4()q [owise] .
eq (SYS1).super.return4()o = (SYS1).return4()o [owise] .

eq (SYS3).area() = (SYS3).area()o,(SYS3).area()q .
eq (SYS5).area() =
    (oval(SYS5)).area()o,(oval(SYS5)).area()q .
eq (SYS3).perimeter() =
    (SYS3).perimeter()o,(SYS3).perimeter()q .
eq (SYS5).perimeter() =
    (oval(SYS5)).perimeter()o,(oval(SYS5)).perimeter()q .

eq (SYS2).super.area() =
    (SYS2).super.area()o,(SYS2).super.area()q .
eq (SYS7).super.area() =
    (oval(SYS7)).super.area()o,(oval(SYS7)).super.area()q .
eq (SYS2).super.perimeter() =
    (SYS2).super.perimeter()o,(SYS2).super.perimeter()q .
eq (SYS7).super.perimeter() =
    (oval(SYS7)).super.perimeter()o,
    (oval(SYS7)).super.perimeter()q .

eq (SYS3).return4() = (SYS3).return4()o,(SYS3).return4()q .
eq (SYS5).return4() =
    (oval(SYS5)).return4()o,(oval(SYS5)).return4()q .

eq (SYS2).super.return4() =
    (SYS2).super.return4()o,(SYS2).super.return4()q .
eq (SYS7).super.return4() = (oval(SYS7)).super.return4()o,
        (oval(SYS7)).super.return4()q .

    eq ((SYS3).side1:=(SYS13)).side1 = SYS13 .
    eq (SYS5).side1:=(SYS13) = (oval(SYS5)).side1:=(SYS13) .
    eq (SYS5).side1 = (oval(SYS5)).side1 .
    eq ((SYS3).side2:=(SYS14)).side1 = (SYS3).side1 .
    eq ((SYS3).number:=(SYS15)).side1 = (SYS3).side1 .
    eq ((SYS3).side2:=(SYS16)).side2 = SYS16 .
    eq (SYS5).side2:=(SYS16) = (oval(SYS5)).side2:=(SYS16) .
    eq (SYS5).side2 = (oval(SYS5)).side2 .
    eq ((SYS3).side1:=(SYS17)).side2 = (SYS3).side2 .
    eq ((SYS3).number:=(SYS18)).side2 = (SYS3).side2 .
    eq ((SYS3).number:=(SYS19)).number = SYS19 .
    eq (SYS5).number:=(SYS19) = (oval(SYS5)).number:=(SYS19) .
    eq (SYS5).number = (oval(SYS5)).number .
    eq ((SYS3).side1:=(SYS20)).number = (SYS3).number .
    eq ((SYS3).side2:=(SYS21)).number = (SYS3).number .

endfm
```

As can be seen this FAS is longer and more complex than the FAS seen in Section 4.2. This is not just due to the addition of new methods, fields and constructors, neither is it just due to the need to import methods, fields, and constructors from Shape. There is a large amount of additional equations and declarations generated to allow us access to the *super* or inherited class' (in this case Shape) original methods and fields, which is important if they have been overriden by the inheriting class (Rectangle).

Throughout the rest of this section we will examine how we model inheritance in the general case and then refer back to the above Rectangle FAS as an illustrative example of each concept we introduce. As before we will not be looking at the declaration of the variables as this is trivial and is only required by the Maude language.

4.3.1 Order of Class Specification Evaluation

In order to create the FAS for a class that imports from another class we need to first traverse the inheritance hierarchy till we reach the topmost class (i.e. the class that does not inherit from any another class).

We then need to generate the FAS for each of this class' methods, fields and constructors as discussed in Section 4.2. However we do not actually need to generate an FAS Maude module for this class but we do need to keep a record of the information generated so that we can import it into the next class in the inheritance chain.

We also need at this stage to note which operators relate to the methods, which ones relate to the constructors, and which to the fields. Therefore when the user generates the FAS, it is important that they keep a record of what each operator relates to.

We then move onto the next class down in the inheritance tree and create an FAS for that class. Then we take the FAS information we generated for the class it inherits from and import it into the new FAS. We will discuss how we do this in the next section.

If this is the last class in the inheritance hierarchy (i.e. the class we wish to actually model) then we output all the FAS information as an FAS Maude module. If not we again categorise the operators and proceed down the inheritance hierachy to the next class.

Fields remained unchanged as it would make no difference to the class to override a field. All we have to do when we import a field is to copy its equations and operators into the inheriting class and of course create any new equations that are needed to search through a class instance's term list for a given field as we saw in section 4.2.4. Therefore we will not refer again to fields in this section.

Constructors are not inherited so we will not be referring to them again either. All we are interested in in this section is modelling how inheritance and methods are translated to an FAS.

In Java when a method is inherited by a class, one of two things can occur:

1. The inherited method is overridden by a new definition in the inheriting class.

2. The inherited method is not overridden by a new definition.

If the method is not overridden then the inherited methods definition is used (as we will see later in our model this means we use the inherited method equations to define the method in the inheriting class). If it is overridden then the new definition is used instead of the inherited definition (we will see later how we make use of a notation called `[owise]` in Maude to allow us to do this).

Whether overridden or not, the original definition of an inherited method can be accessed from the inheriting class via the use of the keyword `super`. For instance if a class inherited a method called `ameth()` from a class, then the original definition could be accessed as follows `super.ameth()`. We will show later how in our model we make use of a serious of operators and linking equations to allow us to model accessing overridden method definitions.

4.3.2 Inheriting Methods

We will look at how we model inherited methods in an FAS, first with a general case where we use `AClass` as the inheriting class and `IClass` as the class that `AClass` will inherit from. We will then use the methods `area` and `return4` to illustrate this. As will be seen `area` is a method that will be overridden and `return4` is a method that is inherited without being overridden.

Firstly we make two copies of all the inherited method's equations. With the first copy we tag the keyword `[owise]` on to the end of each of the method equations. These new equations we will represent the original definitions of the inherited methods. The `[owise]` keyword ensures that we only call the original equations for an inherited method once all other possibilities have been tried. This ensures that if a method is overridden then the overriding definition is executed instead of the original inherited method definition. We need to have the original definitions tagged with `[owise]` to allow us to model methods that may not have been overridden.

$$eq\ term1\ =\ term2\ [owise]\ .$$

In the FAS example this creates the lines:

```
eq (R).area()q = 0 [owise] .
eq (R).return4()q = 4 [owise] .
```

Next we take the inherited class' method operators and tag `super.` onto them between the _. notation and the method name for each reference to the method name. This is done so as the original method definitions are now accessible via the use of the `super` notation. As mentioned earlier if we inherit a method call `ameth()` then the original definition for the method is accessed as follows `super.ameth()`. So if we have a method that was originally in the following format:

$$op\ \_.method()\ :\ IClass\ InputList \longrightarrow\ IClassReturnType\ .$$

This is converted to the following format:

$$op\ \_.super.method()\ :\ IClass\ InputList \rightarrow\ IClassReturnType\ .$$

In the FAS example this produces the following lines:

```
op _.super.area() : Shape  -> ShapeInt .
op _.super.return4() : Shape  -> ShapeInt .
```

Note that we retain the inheriting class' original return types. We will explain how we are able to do this in Section 4.3.3.

If the method is not overridden (i.e. there is no overriding new definition for a method in the inheriting class) then we also need to retain a copy of the original operator for the method but change the class instance input and return types to match the inheriting class' types. As the inherited method will now refer to class instances of the inheriting class type, the class type the method is to be enacted upon as well the return type of state changes to a class instance need to refer to the inheriting class type and not the inherited class type. So if the original operator was of the following format:

$$op\ \_.method()\ :\ IClass\ InputList \rightarrow\ IClassReturnType\ .$$

The new converted copy would be of the following format:

$$op\ \_.method()\ :\ AClass\ InputList \rightarrow\ AClassReturnType\ .$$

In the FAS example `return4` is not overridden so it produces the following extra operator:

```
op _.return4() : Rectangle  -> RectangleInt  .
```

For all inherited methods we also do the same for any operators that handle the tuple query and state change return types for each method as shown in Section 4.2.2. These operators allow us to enact methods belonging to a class on any of the tuple types of that class as these tuples are also valid instances of the class. In our FAS example this accounts for the following lines.

```
op _.super.area() : Shape  -> ShapeInt .
op _.super.area() : ShapeInt  -> ShapeInt .
op _.super.area()o : Shape -> Shape .
op _.super.area()q : Shape -> Int .

op _.return4() : RectangleInt  -> RectangleInt .
op _.return4()o : Rectangle -> Rectangle .
op _.return4()q : Rectangle -> Int .

op _.super.return4() : ShapeInt  -> ShapeInt .
op _.super.return4()o : Shape -> Shape .
op _.super.return4()q : Shape -> Int .
```

All the necessary tuple equations, operators and types for an overridden method's new definition will have already been generated during the initial code generation for the inherited class so we will not discuss them here. The same applies to the new equation definitions for evaluating the method. We will only be showing any new code generated for modelling inheritance in both the general case and the example.

Next we generate equations for each non overridden method, linking the *super* versions of each method's query and state change definitions to its standard version. If a method has not been overridden then by calling a method via the super keyword will be the same as simply calling the method without the keyword. Therefore we define the `super.method()` in terms of the inheriting class' actual `method()` definition. We use the `[owise]` keyword to ensure this equation isn't called if the method has been overridden (if the method has been overridden then we do not want to have `super.method()` calling a `method()` as this will not be the correct behaviour of `super.method` as it will have a different definition).

$$\begin{aligned} eq\ (AClass).super.amethod()q &= (AClass).amethod()q\ [owise]\ . \\ eq\ (AClass).super.amethod()o &= (AClass).amethod()o\ [owise]\ . \end{aligned}$$

We use the `[owise]` keyword to ensure these equations are only called once all other equations have been tried to prevent incorrect behavior. In our FAS example this produces the following lines:

```
eq (SYS1).super.return4()q = (SYS1).return4()q [owise] .
eq (SYS1).super.return4()o = (SYS1).return4()o [owise] .
```

Finally we take the second copy we made of the inheriting class' methods' equations and as we did with the operators we tag `super.` between the `_.` notation and method name for any reference to a method's name in the equation. This will ensure the equations from an inherited class refer to the original definitions of the inherited class. These original definitions are accessed via the super method operator therefore all references to the original methods must be changed to super methods.

In our FAS example this produces the following lines:

```
eq (SYS2).super.area() =
    (SYS2).super.area()o,(SYS2).super.area()q .
eq (SYS7).super.area() =
    (oval(SYS7)).super.area()o,(oval(SYS7)).super.area()q .

eq (SYS2).super.return4() =
    (SYS2).super.return4()o,(SYS2).super.return4()q .
eq (SYS7).super.return4() = (oval(SYS7)).super.return4()o,
    (oval(SYS7)).super.return4()q .
```

4.3.3 Sort, Subsorts, and Hidden Operators

The other aspect of classes that we have to deal with in inheritance are sorts and the hidden operators. Hidden operators are trivial and are copied unchanged from the inherited class into the inheriting class.

Sorts require a bit more work. First of all we have to define a subsort between the inherited class type and the inheriting class type.

$$subsort\ AClass < IClass\ .$$

Therefore the inheriting class is a subsort of the inherited class. In our FAS example this produces the line:

```
subsort Rectangle < Shape .
```

Next if any of the inherited class methods' return types do not have a corresponding tuple in the inheriting class then that type needs to be generated along with any tuple operations and equations for that type. The inheriting class still has to be able to accept these missing tuple types as valid class instances for accessing method and fields from. Therefore they will need to be generated for the inheriting class in addition to all the other tuple types. So for instance if we have a sort tuple type *IClassReturnType* then we need to generate a corresponding sort type *AClassReturnType* if it has not already been generated. This will involve generating all the tuple operators and equations in the same way as we did in section 4.2.2. We will also need to create a subsort for these tuples.

$$subsort\ AClassReturnType\ <\ IClassReturnType\ .$$

The new operations we would need to generate for this are as follows:

$$\begin{aligned} op\ \_,\_ &: AClass\ ReturnType\ -> AClassReturnType\ . \\ op\ oval(\_) &: AClassReturnType\ -> AClass\ . \\ op\ qval(\_) &: AClassReturnType\ -> ReturnType\ . \end{aligned}$$

We will also need the following new equations:

$$\begin{aligned} eq\ oval(aclass, rettype) &= aclass\ . \\ eq\ qval(aclass, rettype) &= rettype\ . \end{aligned}$$

Finally we need to add the linking and extraction equations for the new tuple types to each of the methods in the class.

$$\begin{aligned} &eq\ (aclassrettype).amethod()\ = \\ &\quad (oval(aclassrettype)).amethod()o, (oval(aclassrettype)).amethod()q\ . \end{aligned}$$

In our example there will be no need to generate these new tuple types as they will already exist when we create FAS information to model `Rectangle`'s existing methods.

4.3.4 Multiple Level Inheritance

We also need to be able to model a class that has more than one level in its inheritance tree. We add to our example the `Square` ACS which extends the inheritance chain by inheriting from `Rectangle`.

```
Class Square Extends Rectangle{

    Hidden{
        op ASquare : -> Square .
    }

    Fields{
    }
    Constructors{
        Square : Int .
    }

    Methods{
        op area : -> Int .
        op perimeter : -> Int .
        op setSide : Int -> Int .
    }

    Operations{
    }

    Variables{
        var I : Int .
        var S : Square .
    }

    Equations{
        eq Square(I) = (ASquare).side1:=(I) .
        eq (S).area()q = (S).side1 * (S).side1 .
        eq (S).perimeter()q = (S).side1 * 4 .
        eq (S).setSide(I)q = (S).side1 .
        eq (S).setSide(I)o = (S).side1:=(I) .

    }
}
```

The FAS for this class is as follows:

```
fmod SQUARE is

    protecting BUILDLINK .

    sort Shape .
    sort Rectangle .
    sort Square .
    sort SquareInt .
    sort Squarevoid .
sort RectangleInt .
sort ShapeInt .

subsort SquareInt < RectangleInt .
subsort RectangleInt < ShapeInt .
subsort Rectangle < Shape .
subsort Square < Rectangle .

op Shape() :  -> Shape .
op Rectangle() :  -> Rectangle .
op Rectangle(_,_) : Int Int -> Rectangle .
op Square(_) : Int -> Square .

op AShape : -> Shape .
op ARectangle : -> Rectangle .
op ASquare : -> Square .

op _,_ : Square Int -> SquareInt .
op oval(_) : SquareInt -> Square .
op qval(_) : SquareInt -> Int .
op _,_ : Square void -> Squarevoid .
op oval(_) : Squarevoid -> Square .
op qval(_) : Squarevoid -> void .
op _,_ : Rectangle Int -> RectangleInt .
op oval(_) : RectangleInt -> Rectangle .
op qval(_) : RectangleInt -> Int .
op _,_ : Shape Int -> ShapeInt .
op oval(_) : ShapeInt -> Shape .
op qval(_) : ShapeInt -> Int .
op _.area() : Square  -> SquareInt .
op _.area() : SquareInt  -> SquareInt .
op _.area() : Squarevoid  -> SquareInt .
op _.area()o : Square -> Square .
op _.area()q : Square -> Int .
op _.perimeter() : Square  -> SquareInt .
op _.perimeter() : SquareInt  -> SquareInt .
op _.perimeter() : Squarevoid  -> SquareInt .
op _.perimeter()o : Square -> Square .
op _.perimeter()q : Square -> Int .
op _.setSide(_) : Square  Int -> Squarevoid .
op _.setSide(_) : SquareInt  Int -> Squarevoid .
op _.setSide(_) : Squarevoid  Int -> Squarevoid .
op _.setSide(_)o : Square Int -> Square .
op _.setSide(_)q : Square Int -> void .
op _.area() : Square  -> SquareInt .
op _.area() : SquareInt  -> SquareInt .
op _.area() : Squarevoid  -> SquareInt .
op _.area()o : Square -> Square .
op _.area()q : Square -> Int .
op _.perimeter() : Square  -> SquareInt .
op _.perimeter() : SquareInt  -> SquareInt .
op _.perimeter() : Squarevoid  -> SquareInt .
op _.perimeter()o : Square -> Square .
op _.perimeter()q : Square -> Int .
op _.setSide(_) : Square  Int -> Squarevoid .
op _.setSide(_) : SquareInt  Int -> Squarevoid .
op _.setSide(_) : Squarevoid  Int -> Squarevoid .
op _.setSide(_)o : Square Int -> Square .
op _.setSide(_)q : Square Int -> void .
op _.super.area() : Rectangle  -> RectangleInt .
op _.super.area() : RectangleInt  -> RectangleInt .
op _.super.area()o : Rectangle -> Rectangle .
op _.super.area()q : Rectangle -> Int .
op _.super.perimeter() : Rectangle  -> RectangleInt .
op _.super.perimeter() : RectangleInt  -> RectangleInt .
op _.super.perimeter()o : Rectangle -> Rectangle .
op _.super.perimeter()q : Rectangle -> Int .
op _.super.area() : Square  -> SquareInt .
op _.super.area() : SquareInt  -> SquareInt .
op _.super.area() : Squarevoid  -> SquareInt .
op _.super.area()o : Square -> Square .
op _.super.area()q : Square -> Int .
op _.super.super.area() : Shape  -> ShapeInt .
op _.super.super.area() : ShapeInt  -> ShapeInt .
op _.super.super.area()o : Shape -> Shape .
op _.super.super.area()q : Shape -> Int .
op _.super.perimeter() : Square  -> SquareInt .
op _.super.perimeter() : SquareInt  -> SquareInt .
op _.super.perimeter() : Squarevoid  -> SquareInt .
op _.super.perimeter()o : Square -> Square .
op _.super.perimeter()q : Square -> Int .
op _.super.super.perimeter() : Shape  -> ShapeInt .
op _.super.super.perimeter() : ShapeInt  -> ShapeInt .
op _.super.super.perimeter()o : Shape -> Shape .
op _.super.super.perimeter()q : Shape -> Int .
op _.return4() : Square  -> SquareInt .
op _.return4() : SquareInt  -> SquareInt .
op _.return4() : Squarevoid  -> SquareInt .
op _.return4()o : Square -> Square .
op _.return4()q : Square -> Int .
op _.super.return4() : Rectangle  -> RectangleInt .
op _.super.return4() : RectangleInt  -> RectangleInt .
op _.super.return4()o : Rectangle -> Rectangle .
op _.super.return4()q : Rectangle -> Int .
op _.super.return4() : Square  -> SquareInt .
op _.super.return4() : SquareInt  -> SquareInt .
op _.super.return4() : Squarevoid  -> SquareInt .
op _.super.return4()o : Square -> Square .
op _.super.return4()q : Square -> Int .
op _.super.super.return4() : Shape  -> ShapeInt .
op _.super.super.return4() : ShapeInt  -> ShapeInt .
op _.super.super.return4()o : Shape -> Shape .
op _.super.super.return4()q : Shape -> Int .

op _.side1 : Square -> Int .
op _.side1:=_ : Square Int -> Square .
op _.side1:=_ : SquareInt Int -> Square .
op _.side1 : SquareInt  -> Int .
op _.side1:=_ : Squarevoid Int -> Square .
op _.side1 : Squarevoid  -> Int .
op _.side2 : Square -> Int .
op _.side2:=_ : Square Int -> Square .
op _.side2:=_ : SquareInt Int -> Square .
op _.side2 : SquareInt  -> Int .
op _.side2:=_ : Squarevoid Int -> Square .
op _.side2 : Squarevoid  -> Int .
op _.number : Square -> Int .
op _.number:=_ : Square Int -> Square .
op _.number:=_ : SquareInt Int -> Square .
op _.number : SquareInt  -> Int .
op _.number:=_ : Squarevoid Int -> Square .
op _.number : Squarevoid  -> Int .
```

```
var S0 : Shape .
var SYS0 : Rectangle .
var SYS1 : Shape .
var J : Int .
var R : Rectangle .
var SYS2 : Square .
var SYS3 : Rectangle .
var I : Int .
var S : Square .
var SYS4 : Shape .
var SYS5 : Rectangle .
var SYS6 : Square .
var SYS7 : Int .
var SYS8 : SquareInt .
var SYS9 : void .
var SYS10 : Squarevoid .
var SYS11 : Int .
var SYS12 : RectangleInt .
var SYS13 : Int .
var SYS14 : ShapeInt .
var SYS15 : Int .
var SYS16 : Int .
var SYS17 : Square .
var SYS18 : Square .
var SYS20 : Int .
var SYS21 : Int .
var SYS22 : Int .
var SYS23 : Int .
var SYS24 : Int .
var SYS25 : Int .
var SYS26 : Int .
var SYS27 : Int .
var SYS28 : Int .
var SYS29 : Int .
var SYS30 : Int .

eq Square(I) = (ASquare).side1:=(I) .
eq (S).area()q = (S).side1 * (S).side1 .
eq (S).perimeter()q = (S).side1 * 4 .
eq (S0).setSide(I)o = (S0).side1:=(I) .
eq Shape() = AShape .
eq (S).super.super.area()q = 0 .
eq (S).super.super.perimeter()q = 0 .
eq (SO).super.super.return4()q = 4 .
eq Shape() = AShape [owise] .
eq (S).super.area()q = 0 [owise] .
eq (S).super.perimeter()q = 0 [owise] .
eq (SO).super.return4()q = 4 [owise] .
eq Rectangle() = ARectangle .
eq Rectangle(I,J) =
   ((ARectangle).side1:=(I)).side2:=(J) .
eq (R).super.area()q = (R).side1 * (R).side2 .
eq (R).super.perimeter()q =
   ((R).side1 * 2) + ((R).side2 * 2) .
eq Shape() = AShape [owise] .
eq (S).super.area()q = 0 [owise] .
eq (S).super.perimeter()q = 0 [owise] .
eq (SO).super.return4()q = 4 [owise] .
eq Shape() = AShape [owise]  .
eq (S).area()q = 0 [owise]  .
eq (S).perimeter()q = 0 [owise]  .
eq (SO).return4()q = 4 [owise]  .
eq Rectangle() = ARectangle [owise] .
eq Rectangle(I,J) =
   ((ARectangle).side1:=(I)).side2:=(J) [owise] .
eq (R).area()q = (R).side1 * (R).side2 [owise] .
eq (R).perimeter()q =
   ((R).side1 * 2) + ((R).side2 * 2) [owise] .
eq Square(I) = (ASquare).side1:=(I) .
eq (S).area()q = (S).side1 * (S).side1 .
eq (S).perimeter()q = (S).side1 * 4 .
eq (S).setSide(I)o = (S).side1:=(I) .
eq oval(SYS6,SYS7) = SYS6 .
eq qval(SYS6,SYS7) = SYS7 .
eq oval(SYS6,SYS9) = SYS6 .
eq qval(SYS6,SYS9) = SYS9 .
eq oval(SYS5,SYS11) = SYS5 .
eq qval(SYS5,SYS11) = SYS11 .
eq oval(SYS4,SYS13) = SYS4 .
eq qval(SYS4,SYS13) = SYS13 .
eq (SYS3).super.super.area()q =
   (SYS3).super.area()q [owise] .
```

```
eq (SYS3).super.super.area()o =
    (SYS3).super.area()o [owise] .
eq (SYS3).super.super.perimeter()q =
    (SYS3).super.perimeter()q [owise] .
eq (SYS3).super.super.perimeter()o =
    (SYS3).super.perimeter()o [owise] .
eq (SYS3).super.return4()q = (SYS3).return4()q [owise] .
eq (SYS3).super.return4()o = (SYS3).return4()o [owise] .
eq (SYS3).super.super.return4()q =
    (SYS3).super.return4()q [owise] .
eq (SYS3).super.super.return4()o =
    (SYS3).super.return4()o [owise] .
eq (SYS6).area() = (SYS6).area()o,(SYS6).area()q .
eq (SYS8).area() =
    (oval(SYS8)).area()o,(oval(SYS8)).area()q .
eq (SYS10).area() =
    (oval(SYS10)).area()o,(oval(SYS10)).area()q .
eq (SYS6).perimeter() =
    (SYS6).perimeter()o,(SYS6).perimeter()q .
eq (SYS8).perimeter() =
    (oval(SYS8)).perimeter()o,(oval(SYS8)).perimeter()q .
eq (SYS10).perimeter() =
    (oval(SYS10)).perimeter()o,(oval(SYS10)).perimeter()q .
eq (SYS6).setSide(SYS15) =
    (SYS6).setSide(SYS15)o,(SYS6).setSide(SYS15)q .
eq (SYS8).setSide(SYS15) =
    (oval(SYS8)).setSide(SYS15)o,
    (oval(SYS8)).setSide(SYS15)q .
eq (SYS10).setSide(SYS15) =
    (oval(SYS10)).setSide(SYS15)o,
    (oval(SYS10)).setSide(SYS15)q .
eq (SYS6).area() = (SYS6).area()o,(SYS6).area()q .
eq (SYS8).area() =
    (oval(SYS8)).area()o,(oval(SYS8)).area()q .
eq (SYS10).area() =
    (oval(SYS10)).area()o,(oval(SYS10)).area()q .
eq (SYS6).perimeter() =
    (SYS6).perimeter()o,(SYS6).perimeter()q .
eq (SYS8).perimeter() =
    (oval(SYS8)).perimeter()o,(oval(SYS8)).perimeter()q .
eq (SYS10).perimeter() =
        (oval(SYS10)).perimeter()o,
        (oval(SYS10)).perimeter()q .
    eq (SYS6).setSide(SYS16) =
        (SYS6).setSide(SYS16)o,(SYS6).setSide(SYS16)q .
    eq (SYS8).setSide(SYS16) =
        (oval(SYS8)).setSide(SYS16)o,
        (oval(SYS8)).setSide(SYS16)q .
    eq (SYS10).setSide(SYS16) =
        (oval(SYS10)).setSide(SYS16)o,
        (oval(SYS10)).setSide(SYS16)q .
    eq (SYS5).super.area() =
        (SYS5).super.area()o,(SYS5).super.area()q .
    eq (SYS12).super.area() =
        (oval(SYS12)).super.area()o,
        (oval(SYS12)).super.area()q .
    eq (SYS5).super.perimeter() =
        (SYS5).super.perimeter()o,(SYS5).super.perimeter()q .
    eq (SYS12).super.perimeter() =
        (oval(SYS12)).super.perimeter()o,
        (oval(SYS12)).super.perimeter()q .
    eq (SYS6).super.area() =
        (SYS6).super.area()o,(SYS6).super.area()q .
    eq (SYS8).super.area() =
        (oval(SYS8)).super.area()o,(oval(SYS8)).super.area()q .
    eq (SYS10).super.area() =
        (oval(SYS10)).super.area()o,
        (oval(SYS10)).super.area()q .
    eq (SYS4).super.super.area() =
        (SYS4).super.super.area()o,(SYS4).super.super.area()q .
    eq (SYS14).super.super.area() =
        (oval(SYS14)).super.super.area()o,
        (oval(SYS14)).super.super.area()q .
    eq (SYS6).super.perimeter() =
        (SYS6).super.perimeter()o,(SYS6).super.perimeter()q .
    eq (SYS8).super.perimeter() =
        (oval(SYS8)).super.perimeter()o,
        (oval(SYS8)).super.perimeter()q .
    eq (SYS10).super.perimeter() =
        (oval(SYS10)).super.perimeter()o,
        (oval(SYS10)).super.perimeter()q .
    eq (SYS4).super.super.perimeter() =
        (SYS4).super.super.perimeter()o,
        (SYS4).super.super.perimeter()q .
    eq (SYS14).super.super.perimeter() =
        (oval(SYS14)).super.super.perimeter()o,
        (oval(SYS14)).super.super.perimeter()q .
    eq (SYS6).return4() =
        (SYS6).return4()o,(SYS6).return4()q .
    eq (SYS8).return4() =
        (oval(SYS8)).return4()o,(oval(SYS8)).return4()q .
    eq (SYS10).return4() =
        (oval(SYS10)).return4()o,(oval(SYS10)).return4()q .
    eq (SYS5).super.return4() =
        (SYS5).super.return4()o,(SYS5).super.return4()q .
    eq (SYS12).super.return4() =
        (oval(SYS12)).super.return4()o,
        (oval(SYS12)).super.return4()q .
    eq (SYS6).super.return4() =
        (SYS6).super.return4()o,(SYS6).super.return4()q .
    eq (SYS8).super.return4() =
        (oval(SYS8)).super.return4()o,
        (oval(SYS8)).super.return4()q .
    eq (SYS10).super.return4() =
        (oval(SYS10)).super.return4()o,
        (oval(SYS10)).super.return4()q .
    eq (SYS4).super.super.return4() =
        (SYS4).super.super.return4()o,
        (SYS4).super.super.return4()q .
    eq (SYS14).super.super.return4() =
        (oval(SYS14)).super.super.return4()o,
        (oval(SYS14)).super.super.return4()q .

    eq ((SYS6).number:=(SYS22)).number = SYS22 .
    eq (SYS8).number:=(SYS22) = (oval(SYS8)).number:=(SYS22) .
    eq (SYS8).number = (oval(SYS8)).number .
    eq (SYS10).number:=(SYS22) = (oval(SYS10)).number:=(SYS22) .
    eq (SYS10).number = (oval(SYS10)).number .
    eq ((SYS6).side1:=(SYS23)).number = (SYS6).number .
    eq ((SYS6).side2:=(SYS24)).number = (SYS6).number .
    eq ((SYS6).side1:=(SYS25)).side1 = SYS25 .
    eq (SYS8).side1:=(SYS25) = (oval(SYS8)).side1:=(SYS25) .
    eq (SYS8).side1 = (oval(SYS8)).side1 .
    eq (SYS10).side1:=(SYS25) = (oval(SYS10)).side1:=(SYS25) .
    eq (SYS10).side1 = (oval(SYS10)).side1 .
    eq ((SYS6).number:=(SYS26)).side1 = (SYS6).side1 .
    eq ((SYS6).side2:=(SYS27)).side1 = (SYS6).side1 .
    eq ((SYS6).side2:=(SYS28)).side2 = SYS28 .
    eq (SYS8).side2:=(SYS28) = (oval(SYS8)).side2:=(SYS28) .
    eq (SYS8).side2 = (oval(SYS8)).side2 .
    eq (SYS10).side2:=(SYS28) = (oval(SYS10)).side2:=(SYS28) .
    eq (SYS10).side2 = (oval(SYS10)).side2 .
    eq ((SYS6).number:=(SYS29)).side2 = (SYS6).side2 .
    eq ((SYS6).side1:=(SYS30)).side2 = (SYS6).side2 .


endfm
```

In order to model methods inherited from the `Shape` class (which is inherited through the `Rectangle` class) we need to be able to access `Shape`'s definitions of methods. For instance, for the `area` method we do this by calling the method as `super.super.area`. When we model the `Square` class we will first generate all the equations for `Shape` and tag all the equations operators with information as to which class, methods, and fields they relate to (see Section 4.3.1). We then generate the equations for `Rectangle` including equations to model how it inherits methods from `Shape` as seen in Sections 4.3.2 and 4.3.3, but instead of outputting the FAS information we tag it again and pass it down to the next level, `Square`. At this stage there will exist, for example, an operator `super.area` generated when we modelled the inheritance of `Shape` in `Rectangle`. To model the inheritance of this in `Square` we have to tag on another `super` keyword. The following line of code from the FAS example demonstrates this.

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

We also need to generate the linking equations to link `super.super.area` to `super.area`. The following lines from the FAS example demonstrates this.

```
eq (SYS3).super.super.area()q = (SYS3).super.area()q [owise] .
eq (SYS3).super.super.area()o = (SYS3).super.area()o [owise] .
```

In order to do all this we needed to follow the same procedures shown in Sections 4.3.1 to 4.3.3. The only difference in this case is we will have to keep a reference to which methods are `super` methods. The methods' operators for `Shape` and `Rectangle` and their equations are altered by tagging another `super.` to the name as shown in Section 4.3.1. We do not however need to do anything else with these methods that already have at least one `super` tagged on them except to tag on another `super` keyword and generate the linking equations. All the other methods are handled in exactly the same way as they are dealt with in Section 4.3.1.

This allows us to model classes with many levels of inheritance. However as can be seen from the example, with every level of inheritance this becomes more complex with the generation of many extra operations, equations, sorts, and subsorts. This involves a large amount of book keeping to keep track of where each new equation and operation originates from so as we know how to model it as we proceed down the inheritance chain. In addition we have omitted from our example all the necessary variable declarations for variables that we generate during the conversion process. Maude requires us to declare all variables that we use and also we must make sure that their names do not clash with other already existing variables which again requires more book keeping.

4.4 Interface Tagging, Joining and Flattening

In this final section we will continue our look at the work of [STR03]. We have already examined how they defined an object-oriented *interface definition language* and how our work built on that and extended it. In this section we look at how they model the process of interface *flattening* through the use of *joining* and *tagging*. We will also look at how this relates to how we model inheritance in the FAS.

4.4.1 Interface Flattening

Flattening is an important transformation that takes an interface architecture and flattens it into one stand alone interface. An interface architecture consists of interfaces that import other interfaces which themselves in turn can import other interfaces. Interfaces are collected together into an *Interface Repository.* The process of flattening can be viewed as the process of assembling the individual components of the interface architecture into a single interface that fully describes the whole system. This allows us to simplify a complex architecture by removing the modularity or hierarchy of the system represented by a system's *imports.* An interface architecture is flattened by using two operations called *join* and *tag* on the body of the interfaces.

The operation *join* can be viewed as a form of textual substitution. The operator *tag* can be viewed as a way of maintaining the unique identity of all the components in the interfaces.

The process of flattening is, however, not straightforward. It raises complications which need to be dealt with if the following occur:

1. One interface is dependant on (imports) an interface not in the interface repository.

2. Interfaces do not have unique identifier names (i.e. there is a possibility of name clashes).

3. An interface has already been used at some point during the assembly process (i.e. it has already been imported at an earlier stage).

4. Interfaces are mutually dependant on one another.

[STR03] devote considerable attention to these issues. However in the case of the work in this thesis, we note that none of them will apply to correct Java code hence they do not concern us here.

4.4.2 Joining and Tagging

Tagging is used to record locational information. This is necessary to avoid name clashes of components when we flatten interfaces and also to indicate where each component originated from. [STR03] use lists of names to store locational information. For example

$$tag(addName(m, addName(n, \epsilon Name)), B)$$

This relocates the body B from n to m by using the operator *addName* which adds a name to a list of names. If there is no locational information then the tagging has no effect.

$$tag(\epsilon Name, B) = B$$

[STR03] give two methods for recording location.

- *local context tagging* or *single context tagging* (typically the original location)

Local context tagging can be innermost if it follows the innermost tagging rule.

$$tag(addName(m, addName(n, N)), B) = tag(addName(n, N), B).$$

Or it can be outermost if it follows the outermost tagging rule.

$$tag(addName(n, N), B)) = tag(addName(n, \epsilon Name), B).$$

For example if we have interface i that imports an interface x which itself imports an interface z with body B then the tagging is innermost if body B is tagged as follows:

$$x.B$$

and the tagging is outermost if body B is tagged as follows:

$$z.B$$

The other method for tagging is:

- *global context tagging* or multiple tagging.

Global context tagging means that the axioms for *tag* are not simplified, all of the locational information is maintained. For example if we have interface i that imports an interface x which itself imports an interface z with body B then the tagging is innermost if body B is tagged as follows:

$$z.x.B$$

In the work in this thesis we use a modified form of *Global Context Tagging* which we will show in Section 4.4.3.

Joining is the the process of adding the components in interface bodies together. One of the most important points to consider about this process is what happens when you join a declaration to a body where the declaration already exists in that body. As will be seen later this is equivalent to method overriding in object-oriented programming. The process of tagging resolves many of the difficulties that are caused by adding the same body components from different interfaces. The names of interfaces are tagged to imported body components to identify where they came from.

With the use of joining and tagging [STR03] are able to flatten an interface architecture.

They use the following equation to define the meaning of importing an interface J into an interface I.

$$\begin{aligned}&extend(I,J) = intf(name(I),\\&\quad mrgName(cutName(name(J), imports(I)), imports(J)),\\&\quad join(body(I), body(J)))\end{aligned}$$

Where:

- *cutName* is used to remove the import reference to J in I's list of imports.
- *mrgName* adds all of J's imports into I's import list.
- J's body is then joined with I's.

They then define the flattening of an interface I with respect to an interface J by extending I with the tagged interface J to produce the interface:

$$\begin{aligned}&extend(I, intf(name(J), imports(J),\\&\quad tag(addName(name(J), eName), body(J))))\end{aligned}$$

The interface has:

1. I's name.
2. J's name removed from I's imports and any new imports from J added.
3. J's body tagged with J's name and joined to I's body.

4.4.3 Inheritance and Flattening

We will now examine how the work of [STR03] on flattening relates to our model of inheritance. Java does not allow multiple inheritance, so we do not have to deal with the issues raised by multiple inheritance in our model. Also the other problems that [STR03] consider to do with importing such as cyclicity and missing interfaces (discussed in Section 4.4.1) do not apply to our model as we are only interested in modelling Java classes that can be successfully compiled and thus these problems cannot occur in our model.

When we create the ACS or algebraic interface specifications we do not flatten the specifications at this stage. All we do is state which, if any, other classes a class inherits from.

When we create the FAS, we completely flatten the specification into one class. We do not create FASs for any of the intermediate classes in the inheritance tree. We only generate operations, equations, and sorts for the intermediate classes to be used in the final flattened specification. If a user of our system wishes to generate specifications for these intermediate classes they must model them as the base class of their own inheritance tree.

For example if we have a class B that imports class C and a class A that imports B then if we flatten A we do the following:

1. We follow the inheritance hierarchy of A until we reach the top class in this case C.

2. We then generate all the necessary sorts, operations, and equations for class C and note that they belong to this class. We do not generate an FAS for this class.

3. We move to the next class down in the hierarchy, B

4. We again generate all the necessary sorts, operations, and equations for class B and note that they belong to this class. We do not generate an FAS for this class.

5. We join class C's body (sorts, operators and equations) to B's body by tagging the body of C. We will discuss our method of tagging shortly. We also maintain a record of which methods have been tagged for the reasons discussed in Section 4.3.4.

6. We move down to the base class in the hierarchy, A (the target class).

7. We again generate all the necessary sorts, operations, and equations for class A.

8. We join class B's body (sorts, operators and equations) including all tagged equations and operators from class C to A's body by tagging the body of B.

9. We output all the equations, operators and sorts for class A as the final FAS.

We define the meaning of importing a class J into a class I with the following equation.

$$extend(I, J) = class(name(I), imports(J), join(body(I), tag(body(J)))).$$

This creates a new class with:

- I's name.
- J's imports (note J can only have at most one import due to Java's single inheritance).
- J's tagged body joined with I's body.

We then define the flattening of a class I with respect to class J by extending I with the tagged interface J to produce the interface:

$$extend(I, class(name(J), imports(J), tag(addName(super, eName), body(J)))).$$

The class has:

1. I's name.
2. J's name removed from I's imports and any new imports from J added.
3. J's body tagged with the *super* keyword joined to I's body.

In our model we use our own definition of the *join* function to join the body of an inherited class to the body of an inheriting class. All fields and their defining equations are copied in as they stand from the inherited class. We also need to add extra equations to recognise the newly added fields. Next we copy all the operations and equations for the methods from the inherited class unchanged into the inheriting class but we add a special keyword `[owise]` to the equations to ensure they are not called if they have an overriding definition from the inheriting class.

For example if a class A inherits from class B then if class B has the following equation:

```
eq (S).clear()q = EClass .
```

Then it is converted into the following and joined to A's body:

```
eq (S).clear()o = EClass [owise] .
```

We do not do this for tagged methods which we will discuss below.

We then copy in a second tagged copy of the inherited class method's operations and equations into the inheriting class body. These are used to allow us access to the `super` class' (the inherited class) method definitions. We do this by tagging the key word `super.` to the method's name. This allows us to more closely mimic Java's syntax.

For example if a class A inherits the following operator and equation from class B:

```
op _.clear()o : AClass -> AClass .
eq (S).clear()o = EClass .
```

Then it is converted into the following and joined to *A*'s body:

```
op _.super.clear()o : AClass -> AClass .
eq (S).super.clear()o = EClass .
```

As there is only single inheritance there is no danger of ambiguity as to which inheriting class is being referred to with the keyword `super`. These tagged methods are not copied into a class lower down the hierarchy of an inheritance chain as part of the set of methods copied in without additional tagging. We do pass a copy of them in with an extra `super.` tag added to the method names. So `super.super.amethod` refers to the original method definition of `amethod` from the `super` class of the `super` class of the base class.

So if we have another class *Z* which inherits from class *A* then the *clear* method's operator and equation is converted and joined to *Z*'s body in the following format:

```
op _.super.super.clear()o : AClass -> AClass .
eq (S).super.super.clear()o = EClass .
```

When this has been done all the way through the inheritance chain as seen in Section 4.3 then we have fully flattened the inheritance hierarchy into a single FAS.

4.5 Automatically Building FAS Specifications

In section 3.5 we looked at the program called the Algebraic Specification Generator (ASG) we had implemented to automatically build specifications from Java classes with extra embedded information. In that section we looked particularly at extracting information about the class and generating an ACS from the extracted information. In this section we will focus on the final stages of the program which generate an FAS from the information already extracted. Therefore in this section we will not be discussing extracting information about a class as we have already discussed this in section 3.5. In this section we will give a general overview of the algorithm implemented by the program. The program is too long to show and discuss the complete code so, as in the section 3.5, we will focus on small examples of the code for generating FAS information for the fields of a class. The complete code can be found on the appendix cd and instructions for using the code can be found in appendix A.

4.5.1 Overview Of The Implemented Algorithm for FAS Generation

There are two phases to the ASG for building an FAS specification (assuming the first three stages discussed in section 3.5 have already been performed).

1. Manipulating the extracted data into an appropriate form for the ACS by converting it into appropriate formats and generating all the extra operations and equations required to properly model the class in an FAS.
2. Outputting the newly generated information as an FAS.

Unlike generating an ACS, generating an FAS requires the creation of a lot more extra information and reformatting of the gathered data from a class. All the data has to be reformatted into a complete Maude code. This mainly includes inserting the keyword op in front of method, field, and constructor declarations to make them valid Maude operators. Once this is done the program then generates all the extra operators and equations that have been discussed earlier in this chapter. This means generating accessor and mutator operations for fields and appropriate equations for them as shown in section 4.2.4.

Methods need several new operators and equations as discussed in section 4.2.2. Operators defining each method's query and state change components are generated by the ASG. The ASG also creates special versions of each method operator that return the tuple version of the method's query and state change components. The tuples themselves need are generated as well as operators for joining and extracting the components. For every possible tuple return type, extra operators and equations for each method are generated that will accept each tuple type as the class instance input as each of these tuples is a valid instance type of the overall class. In order to avoid creating multiple copies of a tuple type where multiple methods would return the same tuple type, the ASG scans through all the methods and identifies each unique tuple type that will be required before building them. Constructors are reasonable trivial and only require minor reformatting to turn them into valid Maude operations.

In the case of classes with inheritance, the ASG processes the inheritance chain as discussed in section 4.3.1. To briefly summarise that section, this involves finding the class at the top of the inheritance chain and generating the appropriate data for that class. However at this point the data is not output as an FAS. Instead the program stores the data in special storage structures that contain each bit of data as well as special tags to indicate which class in the inheritance chain it belonged to. The ASG then proceeds down the chain to the next class and does the same again. Once done it merges this class' data with data from the class above and does appropriate conversions and generates extra information neccessary to model the inheritance structure between the two classes (we will discuss this in a bit more detail below). All this information is then again stored in a the special storage structures with extra tags to indicate which class they belong to in the chain and then the ASG proceeds to the next class down in the chain. This is repeated until the base class that is to be modelled is reached at which point, once all the information has been gathered and generated, it is all output as a complete FAS for that class with all the levels of inheritance fully modelled.

When modelling inheritance, the bulk of the specification generation is done on modelling inheritance over methods. This requires the ASG to generate all the extra operators and equations as discussed in section 4.3.2. This includes tagging the keyword `super` onto the front of method names that have been inherited from the class above and altering the appropriate equations so that they call `super.methodname` as opposed to `methodname`. If the method has been overwritten by a class lower down in the inheritance chain then no further extra operators or equations need to be generated as the operator and equations for the overriding method being modelled will supersede the original definitions. However if the method has not been overridden then the ASG generates special linking equations that link the `methodname` operators to the `super.methodname` operators.

Finally extra sorts and subsorts are declared for each inherited class in the inheritance chain as shown in section 4.3.3.

All of the above does not require any extra input from the user. It can all be automatically generated based on the information that would have already been provided to the ASG as discussed in section 3.5. Continuing the example shown in that section we present the FAS code generated for that class by the ASG.

```
fmod ACLASS is

    protecting BUILDLINK .

    sort AClass .
    sort AClassInt .
    sort AClassArray .


    op AClass() :  -> AClass .
    op AAClass : -> AClass .

    op _,_ : AClass Int -> AClassInt .
    op oval(_) : AClassInt -> AClass .
    op qval(_) : AClassInt -> Int .
    op _.return4() : AClass  -> AClassInt .
    op _.return4() : AClassInt  -> AClassInt .
    op _.return4()o : AClass -> AClass .
    op _.return4()q : AClass -> Int .

    op _.afield : AClass -> Int .
    op _.afield:=_ : AClass Int -> AClass .
    op _.afield:=_ : AClassInt Int -> AClass .
    op _.afield : AClassInt  -> Int .

    var A : AClass .
    var SYS0 : AClass .
    var SYS1 : Int .
    var SYS2 : AClassInt .
    var SYS3 : AClass .
    var SYS4 : AClass .
    var SYS6 : Int .
    var SYS7 : Int .
    var SYS8 : Int .

    eq AClass() = AACLass .
    eq (A).return4()q = 4 .
    eq oval(SYS0,SYS1) = SYS0 .
    eq qval(SYS0,SYS1) = SYS1 .
    eq (SYS0).return4() = (SYS0).return4()o,(SYS0).return4()q .
    eq (SYS2).return4() =
        (oval(SYS2)).return4()o,(oval(SYS2)).return4()q .

    eq ((SYS0).afield:=(SYS8)).afield = SYS8 .
    eq (SYS2).afield:=(SYS8) = (oval(SYS2)).afield:=(SYS8) .
    eq (SYS2).afield = (oval(SYS2)).afield .


endfm
```

As can be seen, a large amount of extra information has been generated for this simple example class. An even greater amount of information needs to be generated for larger more complex classes especially those that have several levels of inheritance. Therefore the ASG serves as a powerful tool for automatically generating these large and complex FAS specifications.

4.5.2 Field Example

In this section we will look at how the ASG generates the necessary FAS information for Java fields. We will not be showing the complete code as it is very long but will be focussing on key sections of it. Let us consider the following code.

```
for (int i = 0; i < fields.size(); i++) {

      temp = "op _." + fd.name + " : " + origclass + " -> "
         + fd.type + " .";
      fieldops.add(temp);

      temp = "op _." + fd.name + ":=_ : " + origclass + " "
         + fd.type + " -> " + origclass + " .";
      fieldops.add(temp);

      temp = "eq ((" + classvar + ")." + fd.name + ":=("
         + avar + "))." + fd.name + " = " + avar + " .";
      fieldeqs.add(temp);
```

In the above code, the ASG goes through each field in turn that was discovered earlier as discussed section 3.5. For each field it generates two extra operators. One which is used to change the field's value and one which is used to view the field's value. Each of these is stored as a string in the special `fieldops` storage structure. The last three lines are used to generate the equation that defines the behaviour of the *getting* and *setting* operators. Informally they are defined as $(aclass.afield := avalue).afield = avalue$. Extra operators and equations are also generated to allow each of the classes tuple types to be input as the class instance for accessing the field. However as the code for this is similar to the above code we shall not list it here.

Very little else needs to be done for a field (except for generating any necessary variables which we will not discuss here). The final stage of the ASG concerned with fields is to output the information generated by the above code as an FAS. This is done by the following code.

```
    for (int k = 0; k < fieldops.size(); k++) {
      out.println("\t" + (String) fieldops.get(k));
    }

    for (int i = 0; i < fieldeqs.size(); i++) {
      out.println("\t" + (String) fieldeqs.get(i));
    }
```

This simply prints out the operators and equations that were stored in the special storage structures out to the FAS. The above code therefore generates the following FAS information for the line `public int afield;` from the Java source code (variable declarations have been omitted).

```
  op _.afield : AClass -> Int .
  op _.afield:=_ : AClass Int -> AClass .
  op _.afield:=_ : AClassInt Int -> AClass .
  op _.afield : AClassInt  -> Int .

  eq ((SYS0).afield:=(SYS8)).afield = SYS8 .
  eq (SYS2).afield:=(SYS8) = (oval(SYS2)).afield:=(SYS8) .
  eq (SYS2).afield = (oval(SYS2)).afield .
```

4.6 Sources

The main influence in this chapter was again that of [STR03]. In this chapter their work on interface flattening, joining and tagging influenced the way in which we model inheritance by flattening the inheritance tree of a class into one whole FAS. In this chapter the syntax and style is more heavily influenced by that of the Maude language [SRI05] than it was in the previous chapter. This is obviously due to the fact the specification we created in this chapter was written in the Maude language. We have however tried to keep as close as possible to the syntax and style of the Java language [Sun05g] in order to make the specifications more easier and natural to read for a Java programmer. The actual process of creating the FAS and its format is the work of JB. The code used to automatically build an FAS from a Java class is also the work of JB.

Chapter 5

The Pre-Defined Executable Model System

In Chapters 3 and 4 we showed our methodology for modelling Java classes. This showed a structured set of transformations that would ultimately provide a user with an FAS for a class written in Maude. However there are many features and classes that are part of the Java SDK for which it is not easy to use the transformations to create the FAS. This is because their functionality can be considered unique and special. However all classes in Java are expected to possess and make use of this functionality. In order for our model to include this functionality we pre-define these features and classes and allow user defined FASs access to this pre-defined functionality. We create a unique set of operators and equations for this functionality that cannot be generated using our transformations shown in chapter 4.

In this chapter we will examine how we modelled some of these special features and classes and how we can then generate equations that allow FASs for user defined classes access to this functionality.

We will examine two important features of Java that we have modelled. In Section 5.1 we will look at how we model *arrays*. Every FAS must define equations to model its own type of arrays. We will show how this is modelled in the general case and how we can then systematically generate equations to incorporate arrays in each class we model. In our model, arrays have an internal storage structure and external access notation which provides the user with an interface to arrays that mimics the Java notation. In Section 5.2 we look at how we model the *Reflection* classes. Reflection is used to reason about the structure of a class and class instances. Through methods inherited from the `Object` class all Java classes can provide details about the structure of itself and its methods and fields at runtime. We will show how first we model Reflection and then how we can generate the necessary operators and equations for a class to use the reflection methods. It will be seen that Reflection is quite complex to model and requires the generation of special tuples to store reflection information which can then be used by the Reflection specifications. We are not claiming in this section to have fully modelled the Reflection classes. The aim in this section is to show how we model some of the key concepts of Reflection which could then be expanded upon in future work.

This is by no means a complete and concise model of all of Java's built in functionality, but it does show how we model several key and complex features of this functionality. The purpose of this chapter is to show proof of concept. That is we wish to show how work has been done to implement powerful and important features of Java's large built in functionality and API that can be added to and expanded in the future.

Finally we discuss how our program which can automatically create an FAS from suitably commented Java code was used for testing new functionality that we created for our specification model. As was shown in the chapter 4, the generation of an FAS requires a large amount of book keeping and extra operators and equations. The program we created allows us to automate the generation of all this information as much as possible allowing us to rapidly design and test new functionality in our model.

5.1 Arrays

Arrays are a key data type that is used extensively in Java. In our model we generate equations which define a storage structure for each array type. We create what we call user level equations which allow us to link the more commonly used Java array syntax to our storage structure syntax. We predefine array equations and operators for the primitive data types and any pre-defined classes and generate array equations and operators for the classes that we model using our translation methodology. These array equations and operators only appear at the FAS level of specification.

Below is a set of equations, operators and a sort defining an array in the general case of `AClass`. As usual variable declarations are omitted from this example

```
sort AClassArray .

op null : -> AClass .
op EAClassArray : -> AClassArray .
op _[_] : AClassArray Int -> AClass .
op _[_]:=_ : AClassArray Int AClass -> AClassArray .

op add : AClassArray Int AClass -> AClassArray .
op add : AClassArray AClass -> AClassArray .
op get : AClassArray Int -> AClass .

op _.length : AClassArray -> Int   .

op newAClassArray : Int -> AClassArray .
op new:AClass[_] : Int -> AClassArray .



ceq get(add(ACLASSARRAY,ACLASS),INT) =
    ACLASS if INT = (ACLASSARRAY).length .
ceq get(add(ACLASSARRAY,ACLASS),INT) =
    get(ACLASSARRAY,INT) if INT =/= (ACLASSARRAY).length .

eq (EAClassArray).length = 0 .
eq (add(ACLASSARRAY,ACLASS)).length = (ACLASSARRAY).length + 1 .

ceq add(add(ACLASSARRAY,ACLASS1),INT,ACLASS2) =
    add(ACLASSARRAY,ACLASS2) if INT =
    (ACLASSARRAY).length .
ceq add(add(ACLASSARRAY,ACLASS1),INT,ACLASS2) =
    add(add(ACLASSARRAY,INT,ACLASS2),ACLASS1)
    if INT < (ACLASSARRAY).length .

eq newAClassArray(0) = EAClassArray .
eq newAClassArray(INT) = add(newAClassArray(INT - 1),null) .

eq (ACLASSARRAY)[(INT)] = get(ACLASSARRAY,INT) .
eq ((ACLASSARRAY)[(INT)]:=(ACLASS)) =
    add(ACLASSARRAY,INT,ACLASS) .

eq new:AClass[INT] = newAClassArray(INT) .
```

We will now look at relevant sections of the generic example in turn and show how we create the equivalent for a *Shape* array.

```
sort AClassArray .
op null : -> AClass .
op EAClassArray : -> AClassArray .
```

The first line declares the array sort type for an array of *AClass*. The sort of an array is always the class name with the keyword *Array* suffixed to it. It should be noted that if a user attempts to model a Java class actually called *AClassArray* then this will cause a name clash. Therefore a more appropriate naming structure will need to be used or the sort type should be rewritten if their names clash. However we are only interested in demonstrating the concept of array modelling. The second line is used to declare a constant called *null* of type *AClass*. When an array of any class type is initialised in Java, all of its elements are set to a *null*, to represent a *null* instantiation of the class. Our *null* constant is used to represent the *null* value for that class type. The third line is a constant which is used to represent an empty array (I.E. an array with no elements). This is only neccessary for the internal storage structure representation of the array and is not used by a user of the system.

For an array of class *Shape* this would produce the following lines of code.

```
sort ShapeArray .
op null -> Shape .
op EShapeArray : -> ShapeArray .
```

Next we will look at the internal storage structure of the array. These operators are only used internally in the system and are not used directly by the user.

```
op add : AClassArray Int AClass -> AClassArray .
op add : AClassArray AClass -> AClassArray .
op get : AClassArray Int -> AClass .

op _.length : AClassArray -> Int   .
```

The first three lines of code declare operators that allow us to get and set elements of an array. The first line declares an operator to add an element at a specific position. The second line declares an operator that adds an element to an array at the current position. This operator is used in the construction of an array and defines its structure. The third line declares an operator that will return the value of an element at a given position. The fourth line declares an operator that returns the capacity of the array. This operator is also part of the Java language for arrays and is used both by the user and in the internal structure of the array.

For an array of class *Shape* this produces the following lines of code.

```
op add : ShapeArray Int Shape -> ShapeArray .
op add : ShapeArray Shape -> ShapeArray .
op get : ShapeArray Int -> Shape .
op _.length : ShapeArray -> Int  .
```

Next we have the equations defining these operations.

```
ceq add(add(ACLASSARRAY,ACLASS1),INT,ACLASS2) =
    add(ACLASSARRAY,ACLASS2) if INT = (ACLASSARRAY).length .

ceq add(add(ACLASSARRAY,ACLASS1),INT,ACLASS2) =
    add(add(ACLASSARRAY,INT,ACLASS2),ACLASS1)
    if INT < (ACLASSARRAY).length .
```

These define the *add* operators. The first equation defines what happens if we attempt to add an *AClass* class instance (*ACLASS*2) to an array at position *INT*. If *INT* equals (*ACLASSARRAY*)*.length* then we are at the correct position in the array and we discard the *AClass* class instance called *ACLASS*1. This is stored in the array by the following term *add*(*ACLASSARRAY*, *ACLASS*1). We discard this term and replace it with *add*(*ACLASSARRAY*, *ACLASS*2) which stores the new *AClass* class instance *ACLASS*2 in the array. The second equation is the case where *INT* is less than *ACLASSARRAY.length*. In this case we call *add* again on *ACLASSARRAY* with *INT* and *ACLASS* to continue searching for the element position *INT*. We add *ACLASS*2 to the result of this new *add* call.

For an array of class *Shape* this produces the following lines of code.

```
ceq add(add(SHAPEARRAY,SHAPE1),INT,SHAPE2) =
    add(SHAPEARRAY,SHAPE2) if INT = (SHAPEARRAY).length .

ceq add(add(SHAPEARRAY,SHAPE1),INT,SHAPE2) =
    add(add(SHAPEARRAY,INT,SHAPE2),SHAPE1)
    if INT < (SHAPEARRAY).length .
```

Next we define the *get* operator.

```
ceq get(add(ACLASSARRAY,ACLASS),INT) =
    ACLASS if INT = (ACLASSARRAY).length .

ceq get(add(ACLASSARRAY,ACLASS),INT) =
    get(ACLASSARRAY,INT) if INT =/= (ACLASSARRAY).length .
```

The first equation defines what happens if we attempt to get an element from a position *INT* and *ACLASSARRAY.length* is equal to *INT*. If this is the case then we have the correct element and we return the *ACLASS* at that position. The second equation is the case when *INT* does not equal *ACLASSARRAY.length* in which case we continue searching through the rest of the array.

For an array of class *Shape* this produces the following lines of code.

```
ceq get(add(SHAPEARRAY,SHAPE),INT) =
    SHAPE if INT = (SHAPEARRAY).length .

ceq get(add(SHAPEARRAY,SHAPE),INT) =
    get(SHAPEARRAY,INT) if INT =/= (SHAPEARRAY).length .
```

All of these operators rely on the *length* operator which is defined in the obvious way.

```
eq (EAClassArray).length = 0 .
eq (add(ACLASSARRAY,ACLASS)).length = (ACLASSARRAY).length + 1 .
```

For an array of class *Shape* this produces the following lines of code.

```
eq (EShapeArray).length = 0 .
eq (add(ASHAPEARRAY,SHAPE)).length = (SHAPEARRAY).length + 1 .
```

Next we define an operator and equations for creating and initialising an array of a given size.

```
op newAClassArray : Int -> AClassArray .

eq newAClassArray(0) = EAClassArray .
eq newAClassArray(INT) = add(newAClassArray(INT - 1),null) .
```

The operator *newAClassArray* takes an integer value and creates and initialises an array with the same number of elements as the integer value. The first equation returns the empty array *EAClassArray* if the integer value is zero. The second equation is called if the first fails and adds a *null* element to the result of recursively calling *newAClassArray* on the integer value reduced by one. In Java an array of size *N* indexes from *0* to *N-1*.

For an array of class *Shape* this produces the following lines of code.

```
op newShapeArray : Int -> ShapeArray .

eq newShapeArray(0) = EShapeArray .
eq newShapeArray(INT) = add(newShapeArray(INT - 1),null) .
```

In order for our model to more accurately reflect the Java syntax we also define operators and equations that allow us to use a Java style syntax

```
op _[_] : AClassArray Int -> AClass .
op _[_]:=_ : AClassArray Int AClass -> AClassArray .

op new:AClass[_] : Int -> AClassArray .
```

The first operator allows us to retrieve an element stored at a given index in the array. The second operator adds a new *AClass* to the array at the given position. For example given an array called a then $a[3] := aclass$ adds a class instance *aclass* to an array a at position 3 and $a[3]$ returns the element stored in position 3. We use the := notation instead of the = notation (which is the notation Java uses for assignment) to avoid confusion with the Maude syntax use of =. The third operator takes in an integer value and creates and initialises a new array of a size equal to the integer value.

For an array of class *Shape* this produces the following lines of code.

```
op _[_] : ShapeArray Int -> Shape .
op _[_]:=_ : ShapeArray Int Shape -> ShapeArray .

op new:Shape[_] : Int -> ShapeArray .
```

Finally we define these operators using the following equations:

```
eq (ACLASSARRAY)[(INT)] = get(ACLASSARRAY,INT) .
eq ((ACLASSARRAY)[(INT)]:=(ACLASS)) =
    add(ACLASSARRAY,INT,ACLASS) .

eq new:AClass[INT] = newAClassArray(INT) .
```

The first equation links the Java style syntax array lookup method to the *get* method used in the internal structure. The second equation links the Java style syntax add method to the *add* method used in the internal structure. The third equations links the Java style syntax initialisation method to the *newAClassArray* method used in the internal structure.

For an array of class *Shape* this produces the following lines of code.

```
eq (SHAPEARRAY)[(INT)] = get(SHAPEARRAY,INT) .
eq ((SHAPEARRAY)[(INT)]:=(SHAPE)) =
    add(SHAPEARRAY,INT,SHAPE) .

eq new:Shape[INT] = newShapeArray(INT) .
```

5.2 Reflection

In Java, *Reflection* is a tool used to reason about classes. Its primary functionality is to allow runtime discovery and manipulation of classes, methods and fields. Through special methods inherited from the `Object` class the underlying structure of the class belonging to a class instance can be accessed. This allows us to find out which methods and fields belong to a class and manipulate them (amongst other things).

Reflection on methods allows us to discover a method's input types and return types and to invoke the method. Reflection on fields allows us to discover a field's type and to alter the field's value. Reflection provides additional functionality to what has been discussed here but we will only model a selection of examples of what we consider to be core features of Java Reflection.

In order to model Reflection we have chosen to predefine Java's reflection classes. Although some elements of the class can be automatically generated for Reflection, due to its unique functionality, the core structure of reflection cannot be easily automatically generated like other classes. To accommodate this we predefine the structure of the reflection classes and automatically generate equations for other classes we define to link up with the reflection functionality. We will examine in this section our predefined Java Reflection classes and the automatically generated equations and operators.

5.2.1 The Class Reference Repository

In order to avoid confusion when we refer to the *Class* class throughout this chapter we will use the term *RefClass* to represent the *Class* class. As will be seen later when we define special structures to model Reflection there is a danger in Maude of creating cyclical references and infinite recursion in the evaluation of terms. For instance if a RefClass A refers to a Method instance B then B will also have a reference back to A thus causing a cyclical reference in Maude which could cause it to go into an infinite loop when evaluating terms. In order to avoid this we use strings to represent any references to other RefClasses. We provide a special structure called a *Class Reference Repository* and functions that allow us to use these strings to lookup the actual RefClass. A user does not need to use or have knowledge of the repository as it can be considered to be part of the internal mechanism of the model and the Class Repository's functions and the lookup process is performed by the system automatically. We define a functional module in Maude to represent the Class Reference Repository and its functions.

```
fmod CLASSSTORE  is

    protecting OBJECTALGEBRA .

    sort ClassStore .
    sort ClassRep .

    op (_,_) : String Class -> ClassStore .

    op theClassRep : -> ClassRep .
    op ERep : -> ClassRep .

    op add(_,_) : ClassRep ClassStore -> ClassRep .
    op lookup(_,_) : ClassRep String -> Class .

    vars S S2 : String .
    var R : ClassRep .
    var C C2 : Class .

    eq lookup(ERep,S) = null .
    ceq lookup(add(R,(S,C)),S2) = C if (S <= S2) and
        (S >= S2) .
    eq lookup(add(R,(S,C)),S2) = lookup(R,S2) [owise] .

endfm
```

We will now examine this module in more detail. The line *protecting OBJECTALGEBRA* links this Maude module with another existing Maude module called *OBJECTALGEBRA*. All the sorts for predefined classes need to be declared in the first Maude module as their sort types are often used later in other predefined Maude modules.

Next we define two sorts.

```
    sort ClassStore .
    sort ClassRep .
```

The first sort will be used to define a tuple for storing a class name and the reference to its corresponding RefClass. The second sort is used to define the Class Reference Repository itself.

Now we define a tuple for *ClassStore*

```
    op (_,_) : String Class -> ClassStore .
```

This line defines the structure of *ClassStore*. ClassStore is defined as a tuple of a *String* and a *Class* (called RefClass in this discussion). A Class Reference Repository entry is therefore a RefClass with a unique string to identify it.

Next we define constants for *ClassRep*

```
    op theClassRep : -> ClassRep .
    op ERep : -> ClassRep .
```

The first line declares the actual Repository which we will add ClassStores to and look them up from. The second line defines an empty Class Reference Repository called *ERep*.

Next we declare operators on ClassRep.

```
    op add(_,_) : ClassRep ClassStore -> ClassRep .
    op lookup(_,_) : ClassRep String -> Class .
```

The first line defines an operator *add* which takes in a Repository and a ClassStore and adds the ClassStore to the Repository. The second line defines an operator *lookup* that takes in a repository and a String and lookups up the the String in the Repository and returns the corresponding RefClass.

Next we declare variables.

```
    vars S S2 : String .
    var R : ClassRep .
    var C C2 : Class .
```

These variables are used in the equations defining the operators discussed above. They need to be declared as Maude requires all variables used to be explicitly declared. We will ignore the variables throughout the rest of this chapter as their declaration is trivial.

Finally we define equationally the *add* and *lookup* functions.

```
eq lookup(ERep,S) = null .
ceq lookup(add(R,(S,C)),S2) = C if (S <= S2) and
    (S >= S2) .
eq lookup(add(R,(S,C)),S2) = lookup(R,S2) [owise] .
```

Informally the first line says that if we lookup a String in an empty repository then we return a *null* class instance. It should be noted that in our model this should never occur as the model will never lookup a string that does not exist in the repository. However we have included the condition for completeness. The second line says that if we lookup a String *S2* in a repository where *(S,C)* is the head element and *R* is the tail then we return the RefClass *C* if *S* is the same string lexigraphically as *S2*, the search string (Maude does not provide an equality operator for strings so we have to check that *S* is both less than or equal to and greater than or equal to *S2*). The third line of code is only used if the other two equations fail. It discards the head element *(S,C)* of a repository and continues searching for String *S2* on the tail *R*.

In order to use the class repository a list of class name strings together with the name of each class' corresponding RefClass instant need to be stored for later use. These are then used once all the other modelling stages have been performed to create the class repository by adding the following equation to a Maude module.

$$eq\ theClassRep = add(\ldots(add(ERep, AClass1),\ldots), AClassN)\ .$$

5.2.2 The *Class* Class

Every class in Java has a corresponding RefClass instance. The RefClass provides a large functionality [Sun05c]. However we will only define part of the functionality as examples of some of the functionality we consider to be core to Reflection. For instance some of RefClass' methods can return a list of the *private* and *protected* class members. As our model abstracts away from internal representation of classes we only model the public interface. Therefore there are no *private* or *protected* methods in our model and so we are unable to model the methods that retrieve these members.

We will model three of the functions in the RefClass: *getMethods*, *getFields*, and *getName*.

- The method *getMethods* returns an array of *Method* class instances that represent methods that belong to the class represented by the given RefClass instance.
- The method *getFields* returns an array of *Field* class instances that represents fields that belong to a class.
- The method *getName* returns the name of a class represented by the RefClass instance as a string.

We define a functional Maude module called *CLASS* to represent and define the structure of the class *Class* (RefClass).

```
fmod CLASS is

    protecting CLASSSTORE .

    op (_,_,_) : String MethodArray FieldArray -> Class .

    op _.getFields() : Class -> FieldArray .
    op _.getMethods() : Class -> MethodArray .
    op _.getName() : Class -> String .

    var F : FieldArray .
    var M : MethodArray .
    var S : String .

    eq (S,M,F).getName() = S .
    eq (S,M,F).getFields() = F .
    eq (S,M,F).getMethods() = M .


endfm
```

Note the above code does not include operators and equations for defining an array of *Class*es which have been omitted to simplify the example. Also the declarations of the sort type for RefClass is not shown here as it is declared in the functional Maude module *OBJECTALGEBRA* along with a subsort.

```
sort Class .
subsort Class < Object .
```

The line `protecting OBJECTALGEBRA` is used to link up with other predefined Maude modules that are used in the internal representation of classes.
The next lines defines a tuple.

```
op (_,_,_) : String MethodArray FieldArray -> Class .
```

This allows us to define a tuple which stores the structure of a RefClass. We have decided that a RefClass will consist of:

1. a String name,
2. an array of *Method* class instances,
3. an array of *Field* class instances.

Next we declare the operators for the RefClass' methods.

```
op _.getFields() : Class -> FieldArray .
op _.getMethods() : Class -> MethodArray .
op _.getName() : Class -> String .
```

These define operators for *getFields*, *getMethods*, and *getName* respectively.
Finally we define the operators with equations.

```
eq (S,M,F).getName() = S .
eq (S,M,F).getFields() = F .
eq (S,M,F).getMethods() = M .
```

These are projection functions that project out parts of the RefClass tuple. The first line defines *getName* by projecting out the String component of the RefClass tuple. The second equation defines *getFields* by projecting out the array of *Field* class instances component from the RefClass tuple. The final equation defines *getMethods* by projecting out the array of *Method* class instances component from the RefClass tuple.

5.2.3 The Field Class

The *Field* class is used to define the structure of a field and functions that can be performed on it. Every field in every class in Java has a corresponding *Field* class instance. We define three of the *Field* methods: *getName*, *getType*, and *getDeclaringClass* where:

- The method *getName* returns the name of the field as a String.
- The method *getType* returns the RefClass representing the fields type. If this is a primitive type such as an *int* then this returns the wrapper class version (e.g. *Integer*).
- The method *getDeclaringClass* returns the RefClass that represents the class that declared the field.

We define a Maude functional module to represent *Field* called *FIELD*.

```
fmod FIELD is

    protecting CLASS .

    subsort Field < Object .

    op (_,_,_) : String String String -> Field .

    op _.getName() : Field -> String .
    op _.getType() : Field -> Class .
    op _.getDeclaringClass() : Field -> Class .

    var S S2 : String .
    var C : String .

    eq (S,C,S2).getName() = S .
    eq (S,C,S2).getType() = lookup(theClassRep,C) .
    eq (S,C,S2).getDeclaringClass() = lookup(theClassRep,S2) .


endfm
```

Again for simplicity we have omitted the operators and equations defining arrays of fields. The sort type for *Field* is already declared in *OBJECTALGEBRA* as

```
    sort Field .
```

The first line for the functional module defines the subsort.

```
    subsort Field < Object .
```

This declares the sort *Field* as a subsort of *Object*. That is *Field* inherits from *Object* and therefore *Field* is a less general type of *Object*.

The next line defines a tuple.

```
    op (_,_,_) : String String String -> Field .
```

This defines *Field* as being a tuple consisting of three Strings. The first String is the fields name, the second string is the name of the *RefClass* that represents the field's type and the third string is the *RefClass* that represents the class that declared the field. We use strings to represent the *RefClasses* to avoid cyclicle references in Maude as discussed earlier. As will be seen below, we use these strings together with the Class Reference Repository and its *lookup* function to obtain the actual *RefClasses*.

Next we declare the operators for *Field*'s methods.

```
    op _.getName() : Field -> String .
    op _.getType() : Field -> Class .
    op _.getDeclaringClass() : Field -> Class .
```

These define operators for *getName*, *getType*, and *getDeclaringClass* respectively.

Next we define the semantics of these operations through the following equations:

```
eq (S,C,S2).getName() = S .
eq (S,C,S2).getType() = lookup(theClassRep,C) .
eq (S,C,S2).getDeclaringClass() = lookup(theClassRep,S2) .
```

The first equations defines *getName* as a projection function that projects out the name string in the *Field* tuple. The second equation defines *getType*. It passes the type string to the class repository lookup function to lookup the RefClass that matches the string passed to it and returns it as the result of *getType*. The third equations defines *getDeclaringClass* in exactly the same way but passes the declaring class string part of the tuple to the lookup function.

5.2.4 The Method Class

The *Method* class is used to define the structure of a method and the functions that can be performed on it. Every method in every class in Java has a corresponding *Method* class instance. We define five of the *Method* methods: *getName*, *getDeclaringClass*, *getReturnType*, *getParameterTypes*, and *invoke* where:

- The method *getName* returns the name of the method as a String.
- The method *getDeclaringClass* returns the RefClass that represents the class that declared the method.
- The method *getReturnType* returns the RefClass representing the method's return type. If this is a primitive type such as an *int* then this returns the wrapper class version (e.g. *Integer*).
- The method *getParameterTypes* returns an array of RefClass representing the method's input parameter types. If any are a primitive type such as an *int* then this returns the wrapper class version (e.g. *Integer*).
- The method *invoke* allows us to invoke the method represented by the *Method* instance which we will discuss in more detail below.

The method *invoke* allows a user to invoke the underlying method represented by a *Method* class instance. It has two parameters. The first parameter is a class instance of the class that the method belongs to on which we want to invoke the method. This is passed in as the *Object* class. This is valid as all other classes inherit from *Object* and are therefore subtypes of *Object*. When the actual method is invoked the class instance is passed to it. If the class instance then turns out to be the wrong subtype Java would throw an *exception*. However our model does not include exceptions so our Maude code will be unable to evaluate a call to *invoke* if the wrong type is passed (see section 6.7 for more on the exception problem). The second input parameter is an array of *Object*. These must be in the correct order and be of the correct subtypes of *Object* as the input parameters of the method to be invoked. If any of the input parameters are primitive types then they are passed in the array as wrapper classes and the wrapper class value extracted when it is passed to the actual method. Again exceptions are thrown if the parameters in the array are incorrect for the method to be invoked. As stated above we do not model exceptions therefore *invoke* will fail to evaluate if they are not correct. The method invoke returns the result of the method as an *Object* instance. If the method returns a primitive then it is returned as the wrapper class instance version (returned as the more general *Object* type). If it does not return anything (the return type is *void*) then *invoke* returns *null*.

We define a functional Maude module called *METHOD* to represent and define the structure of the class *Method*.

```
fmod METHOD is

    protecting FIELD .

    subsort Method < Object .

    op (_,_,_,_) : String StringArray String String -> Method .

    op _.getName() : Method -> String .
    op _.getDeclaringClass() : Method -> Class .
    op _.getReturnType() : Method -> Class .
    op _.getParameterTypes() : Method -> ClassArray .
    op _.invoke(_,_) : Method Object ObjectArray -> Object .

    op getParameterTypesAux(_,_,_) :
        StringArray Int ClassArray -> ClassArray .


    var S S2 : String .
    var C : String .
    var CA : StringArray .
    var RESULT : ClassArray .
    var A : MethodArray .
    vars I J K : Int .
    vars X Y : Method .

    eq (S,CA,C,S2).getName() = S .
    eq (S,CA,C,S2).getDeclaringClass() =
        lookup(theClassRep,S2) .
    eq (S,CA,C,S2).getReturnType() = lookup(theClassRep,C) .
    eq (S,CA,C,S2).getParameterTypes() =
        getParameterTypesAux(CA,0,EClassArray) .

    ceq getParameterTypesAux(CA,I,RESULT) = RESULT
        if I == (CA).length .
    ceq getParameterTypesAux(CA,I,RESULT) =
        getParameterTypesAux(CA,I + 1,
        add(RESULT,lookup(theClassRep,CA[I])))
        if I =/= (CA).length .

endfm
```

Again for simplicity we have omitted the operators and equations defining arrays of methods. The sort type for *Method* is already declared in *OBJECTALGEBRA* as

```
    sort Method .
```

The first line for the functional module defines the subsort.

```
    subsort Method < Object .
```

The next line defines a tuple.

```
op (_,_,_,_) : String StringArray String String -> Method .
```

This defines *Method* as being a tuple consisting of a String followed by a String array and then two more Strings. The first string is the method's name, the string array is an array of the names of the RefClasses that represents the methods's input types, the third string is the RefClass that represents the class that declared the method, and the fourth string is the name of the RefClass the represents the method's return type . We use strings to represent the RefClasses to avoid cyclical references in Maude as discussed earlier. As will be seen below, we use these strings together with the Class Reference Repository and its *lookup* function to obtain the actual RefClasses.

Next we define operators for *Method*'s own methods.

```
op _.getName() : Method -> String .
op _.getDeclaringClass() : Method -> Class .
op _.getReturnType() : Method -> Class .
op _.getParameterTypes() : Method -> ClassArray .
op _.invoke(_,_) : Method Object ObjectArray -> Object .
```

These define operators for *getName*, *getDeclaringClass*, *getReturnType*, *getParameterTypes*, and *invoke* respectively. It should be noted that we will not give any equation defining *invoke* in the *METHOD* module. Each *Method* instance has its own definition for invoke. This will be defined for each method when a class is modelled, which we will look at in Section 5.2.5. These *invoke* equations can be generated automatically.

Finally we give the equations defining these operators.

```
eq (S,CA,C,S2).getName() = S .
eq (S,CA,C,S2).getDeclaringClass() =
    lookup(theClassRep,S2) .
eq (S,CA,C,S2).getReturnType() = lookup(theClassRep,C) .
eq (S,CA,C,S2).getParameterTypes() =
    getParameterTypesAux(CA,0,EClassArray) .
```

The first equation defines *getName* as a projection function that extracts the name string from the *Method* tuple. The next equation defines *getDeclaringClass* by extracting the declaring class string from the tuple and returning the RefClass that the string represents by passing the it to the class repository lookup function. The next equation does the same but passes the return type string instead to the class repository lookup function. The final equation defines *getParameterTypes* by extracting the string array from the *Method* tuple and passing it together with an integer 0 and an empty class array to a function called *getParameterTypesAux.*

Finally we define *getParameterTypesAux.* First we declare its operator.

```
op getParameterTypesAux(_,_,_) :
    StringArray Int ClassArray -> ClassArray .
```

This defines *getParameterTypesAux* as a function that takes in a string array, an integer, a RefClass array, and returns a RefClass array. Informally its purpose is to take in an array of strings that represent RefClass names and convert this into a RefClass array by looking up in the class repository the RefClass for each string in the string array in turn.

Finally we define the conditional equations for *getParameterTypesAux*

```
ceq getParameterTypesAux(CA,I,RESULT) = RESULT
    if I == (CA).length .
ceq getParameterTypesAux(CA,I,RESULT) =
    getParameterTypesAux(CA,I + 1,
    add(RESULT,lookup(theClassRep,CA[I])))
    if I =/= (CA).length .
```

The first equation is the base case and returns the RefClass Array parameter (`RESULT`). It is only used if the integer `I` is the same size as the length of the string array (`CA`). That is, it is only used if we have already looked up each string in the string array. The second equation passes the string in the string array (`CA`) at the `I`th position to the class repository lookup function. The RefClass this returns is added to the RefClass array (`RESULT`) and *getParameterTypesAux* is called recursively.

5.2.5 Modelling Reflection on User Defined Classes

When a user wishes to model their own defined classes then for every method and field in their class they will need to define *Method* and *Field* instances. These can then be used to define the *RefClass* instance for their class which will also need to be added to the class reference repository. In addition, for every method modelled they will need to provide equations for the *invoke* method definiton for each method in their class. This can all be done automatically. We will show here the operations and equations that needs to be generated for the Reflection.

First we will look at how we create a *Field* instance for an arbitrary field.

Suppose in a user defined Java class which we will class MyClass we declare the following field

```
public String mystr;
```

We will need to define the *Field* instance for this field. First we need to define a constant operator for the field. We have chosen to follow the following naming pattern.

- The field's name
- The name of the class it belongs to
- The word 'Field'.

So this creates in the general case a name of the following format

```
afieldnameAClassField
```

This is used to declare the following operator.

```
op afieldnameAClassField : -> Field .
```

In our example this produces the name

```
mystrMyClassField
```

We use this to declare the following operator.

```
op mystrMyClassField : -> Field .
```

It should be noted that there is the potential for name clashes if another component in a Java class is called `mystrMyClassField` and a mechanism for tracking this and altering the name if this occurs will be needed.

Next we define the operator as a tuple using an equation. Recall a `Field` tuple consists of three strings.

- The name of the field.
- The name of the declaring class.
- The name of the class representing the field's type.

In the general case the equation that defines this operator is as follows

```
eq afieldnameAClassField = ("afieldname","AClass","TypeClass") .
```

In our example this produces the following equation.

```
eq mystrMyClass = ("mystr","MyClass","String");
```

Next we look at how we create a *Method* instance for an arbitrary method.

Suppose in our Java class `MyClass` we have a method of the following format.

```
public int mymeth(Integer ic,int ip)
```

We are not interested in the implementation of the method itself here. We need to define the *Method* instance for this method. First we define a constant operator to represent the *Method* instance. We have chosen the following naming pattern.

- The method's name.
- The name of the method's declaring class.
- The keyword 'Method'.

In the general case this produces the following name:

```
amethodAClassMethod
```

This is used to create the following operator:

```
op amethodAClassMethod : -> Method .
```

In our example this produces the following name:

```
mymethMyClassMethod
```

This is used to create the following operator:

```
op mymethMyClassMethod : -> Method .
```

It should be noted that there is the potential for name clashes if another component in a Java class is called `mymethMyClassField` and a mechanism for tracking this and altering the name if this occurs will be needed.

Next we define the operator as tuple using an equation. Recall a `Method` tuple consists of four components.

- The String name of the method.
- An array of String names of the classes that represent the method's input parameters. Note that the string name of a class instance that is any subtype of *Object* is allowed in the String array.
- The String name of the class representing the methods's return type.

- The String name of the declaring class.

In the general case the equation that defines this operator is as follows:

```
eq amethAClassMethod = ("ameth",
      add(...(add(add(EStringArray,"InputClass1"),
      "InputClass2"),...),InputClassN),
      "RetTypeClass","AClass") .
```

In our example this produces the following equation.

```
eq mymethMyClassMethod =("mymeth",
    add(add(EStringArray,"Integer"),"Integer"),"
    Integer","MyClass") .
```

Note that both the primitive integer input and return types are represented by its wrapper class (`Integer`). This is true for all primitive return types.

We also need to provide the defining equation for *Method*'s invoke operator for this method.

In the general case this produces the following:

```
eq ("ameth",ArrayofStrings,"RetType","AClass").invoke(O,I) =
    (O).ameth(I[0],I[1],...,I[N-1]) .
```

If any of the input types are primitive types then they will need to be unwrapped when passed to the actual method. For example:

```
eq ("ameth2",ArrayofStrings,"RetType","AClass").invoke(O,I) =
    (O).ameth((I[0]).getValue()) .
```

Also if the return type of a method is a primitive type then it needs to be wrapped before it is returned to the *invoke* method. For example:

```
eq ("ameth3",ArrayofStrings,"RetType","AClass").invoke(O,I) =
    WrapperClass((O).mymeth(I[0])) .
```

In our example this produces the following equation.

```
eq ("mymeth",add(add(EStringArray,"Integer"),"Integer")
    ,"Integer","MyClass").invoke(O,I) =
    Integer((O).mymeth(I[0],I[1].getValue()))
```

Next we need to produce the RefClass for *MyClass* that will contain all the *Method* and *Field* instances we have created for the class. First we need to define a constant operator for the class. We have adopted the following naming convention:

- The name of the class.
- The keyword 'Class'.

In the general case this produces the following name:

```
AClassClass
```

This produces the following operator:

```
op AClassClass : -> Class .
```

In our example this produces the following name:

```
MyClassClass
```

We use this name to produce the following operator:

```
op MyClassClass : -> Class .
```

It should be noted that there is the potential for name clashes if another component in a Java class is called `MyClassClass` and a mechanism for tracking this and altering the name if this occurs will be needed.

Next we define the operator as tuple using an equation. Recall a `Class` tuple consists of three components.

- The String name of the class.
- An array of the class' *Methods*.
- An array of the class' *Fields*.

In the general case the equation that defines this operator is as follows

```
eq AClassClass = ("AClass",add(...(add(add(EMethodArray,
    ameth1AClassMethod),ameth2AClassMethod),...),
    amethNAClassMethod),add(...(add(add(EFieldArray,
    afield1AClassField),afield2AClassField),...),
    afieldNAClassField)) .
```

In our example if we assume that *MyClass* consists only of the field and method we defined earlier then we produce the following equation.

```
eq MyClassClass = ("MyClass,add(EMethodArray,mymeth),add(EFieldArray,mystr)) .
```

Finally we need to add this together with any other RefClasses (which will be stored in a special file for future reference) to *theClassRep* to build the Class Reference Repository.

This will produce the following equation.

```
eq theClassRep = add(...(add(add(ERep,("AClass1",
    AClass1Class)),("AClass2",AClass2Class)),...),
    ("MyClass",MyClassClass)) .
```

5.3 The Automated Conversion Program

In this section we will examine the tool that we used to assist the development of our model. The tool allowed us to add and test new functionality to our model and many of the important concepts of our specification process were trialed using this tool. We have called the tool the Algebraic Specification Generator (ASG).

The ASG allows the automatic generation of all the extra equations needed to fully define the full algebraic specification (FAS) of the class as discussed in chapter 4. This allowed us to quickly test new ideas without having to write out all the operators and equations by hand. This leads to another possible use for the tool. The process of taking an algebraic class specification (ACS) in the format described in chapter 3 and creating the FAS with all the extra information needed to define the complete semantics of a class as shown in chapter 4 can be very complex.

Many extra operation and equations have to be created in order for the specification to be fully defined. This becomes even more complex when the class to be specified is part of an inheritance hierarchy as all the classes in the inheritance chain have to be specified and then equations generated that handle access to super class methods and deal with inherited method semantics. The ASG can automate this process allowing the user to only have to enter a minimal amount of specification information themselves.

However the ASG was not originally designed for this purpose and was aimed at allowing us to define and test new specification ideas in our model. Therefore more work would need to be done in order to make it more usable and robust. For instance the tool is not foolproof. The ASG is vulnerable to slight syntactic variations in the layout of the method equations in the *Javadoc* comments such as the presence of unexpected whitespace characters. This would need to be improved if it were to be used for the purpose of generating specifications for user defined classes.

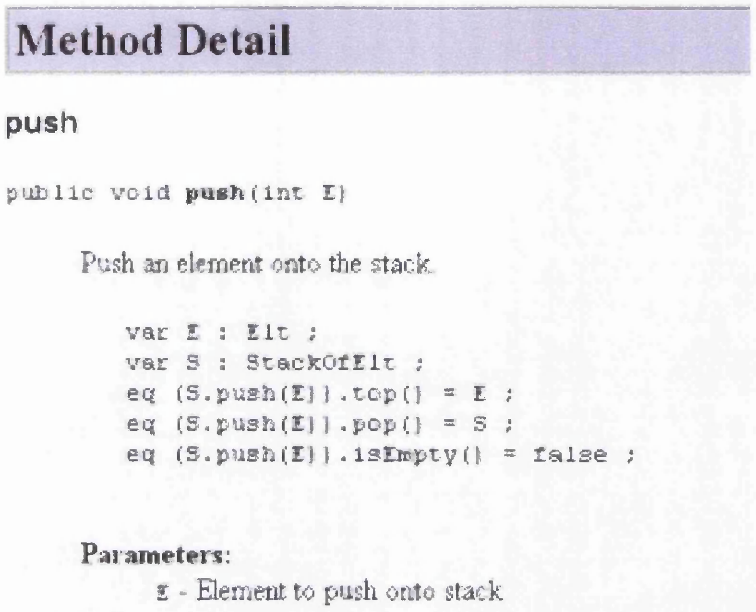
```
/**
 *
 * @return boolean
 * <code>
 * var S : Stack .
 * var O : Object .
 * eq (AStack).empty()q = true .
 * eq ((S).push(O)o).empty()q = false .
 * </code>
 */
public boolean empty(){
```

The above example shows how we embed formal specification information within Javadoc comments. Javadoc comments are used by the Sun-supplied *javadoc* tool to create API HTML documentation as seen in [Sun05g]. We embed our equations for the semantics of methods and constructors in the Javadoc comments in the source file using HTML tags that our program recognises. This means that when the *javadoc* tool is run on source code, equations (and related information) are copied into the API HTML documentation. Hence the defining equations become part of the javadoc documentation. For instance the above example could be displayed in the documentation in a format similar to figure 5.1.

At present the specification information is just added into the documentation unformatted but in future work we would hope to add more tags to our embedded equations that Javadoc recognises so that the equations are added into the documentation with suitable formatting and headings.

The ASG reads this embedded information and extracts it to generate a fully formed FAS. The ASG takes as input the original source code *.java* file, extracts the embedded specification information and combines it with other information that it obtains from the compiled class itself. In this way as can be seen from the example above a user only has to provide the bare minimum of formal information for a class and the tool is able to generate all the other equations and operators for the FAS. The FAS is output as Maude code to allow us to be able to execute specifications. However the ASG need not be limited to Maude output and could be converted to other forms of specification language output if desired.

We use the tags `<code>` and `</code>` to encapsulate our specification declarations which the ASG will extract. Embedded within these tags are equations and variable declarations for the equations that define the method



Method Detail

push

```
public void push(int I)
```

Push an element onto the stack.

```
var I : Elt ;
var S : StackOfElt ;
eq (S.push(I)).top() = I ;
eq (S.push(I)).pop() = S ;
eq (S.push(I)).isEmpty() = false ;
```

Parameters:
I - Element to push onto stack

Figure 5.1: Formal API Documentation.

or constructor (in the earlier example, a method called *empty*). The program will associate these declarations with the *empty* method. We embed equations and variable declarations for the constructors in exactly the same way. We will not discuss the meaning or format of the equations and variable declarations themselves as this is discussed in more detail in chapters 3 and 4 of this thesis.

In addition to *code* tags, we allow *hidden* tags as shown in the following example.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 *
 * <hidden> op AStack : -> Stack .</hidden>
 */
```

```
public class Stack {
```

In the above example we embed additional specification information in the Javadoc comments that appear at the top of the class describing class wide information. We embed this information within `<hidden>` and `</hidden>` tags. Any declarations within these tags are copied into the FAS unchanged. This is useful if we want to declare special operators or equations that are used in many method specification equations throughout the class.

A common declaration as seen above is to declare an operator to symbolise an initial or empty state of an instance of the class. The example above declares an operator `AStack` which is used to denote an empty stack and as will be seen in chapter 6 it is used by many method equations for the *Stack* class. It would be possible to generate `AStack` automatically. However other information may still be needed that can not so easily be automatically generated.

In Maude it is required that all variables are declared. It also a requirement that each variable name is unique. All of the variable declarations are checked by the ASG to make sure there are no name clashes. If there is name clash for a variable and they are both the same type then one of the variable declarations is simply deleted. If they are not the same type and they clash then the name of one the variables is changed along with any reference to it in the corresponding method/constructor equations. The same is not true of equations and it is acceptable to have duplicate equations in Maude. Therefore the ASG does not need to check for duplicate equations.

The ASG uses Java Reflection [McC98] to discover the interface information for the class such as class name, method name, field name, field type, method input names and types etc (the use of Reflection in the ASG was a motiviation for modelling the Java Reflection classes). This information is used to create extra operators and equations as described in Chapter 4 to create the FAS . If the class being specified is part of an inheritance chain then the ASG will find the class at the top of that inheritance chain and automatically specify that class and then move down the chain and specify the next class. It will then combine that specification with the specification from the class above and generate appropriate operations and equations to model the super class methods and inheritance properties. This is then repeated with the next class down and so on till the ASG has specified all the way down to the class we have asked it to specify.

Finally we would like to point out that many of the algebraic specification examples in the next chapter were generated using this tool.

5.4 Sources

The majority of the work in this chapter makes use of the modelling techniques we defined in Chapters 3 and 4 and is the work of JB. We have shown how to model Java arrays [Sun05f] and Java Reflection [McC98] and have used Java's own definitions of these as the basis for our modelling of them. Also we have used Java's Reflection API in our automated conversion tool to discover the interface of the class we are specifying. The idea of embedding formal specifications in Java comments is similar to that of [JML05]. The actual ASG program is the work of JB.

Chapter 6

Examples Of Class Specifications

In this chapter will look at a series of examples that illustrate all the concepts of our modelling technique. These examples will not only show the capabilities of our model but will also be used to demonstrate the current limitations.

The majority of these examples are taken from the Java v1.4.2 API. We have chosen the API library for several reasons. Firstly it provides a set of complete and relatively well documented classes. Also in order to be able to model our own classes we will need to make use of many of Java's internal classes and will therefore need to model them. In order to determine the functionality of the classes that we are modelling we have referred to the API's informal documentation [Sun05g]. In many cases this has demonstrated the inadequacies of informal documentation as the Java documentation was often ambiguous and confusing. When this occurred we were forced to refer back to the original source code for the API in order to try and identify the actual functionality. Although in some cases this proved useful, in others the code was hard to interpret. This shows why models such as ours would be useful in program development. Throughout this chapter we will not always present the full example and it's corresponding FAS, but focus on key points from the examples. Also in the Java examples we have removed the original code from the method bodies as this is not of importance in this chapter. Only a few of the classes will model reflection in the FAS. This is again to reduce the size of the examples. Finally as our model is unable to model exceptions as will be discussed in section 6.7, we have removed all references to exceptions from the code examples.

This chapter will be split into two parts. Sections 6.1 to 6.6 will focus on a set of examples that represent successful modelling of Java classes using our specification methods. There are however some elements of the original classes that proved difficult or impossible to specify using our current modelling techniques. In these instance we will indicate and discuss these problems. In general though, this section demonstrates our modelling process functioning and working on a variety of examples. The examples will cover everything from basic class specification, to classes with inheritance and Reflection. Section 6.7 will examine specific cases from other classes which we know we cannot model. We will look at why we cannot specify them and suggest how future work could improve our model to accommodate these problems. We feel that with time all the problems discussed in this section can be solved and incorporated into our specification techniques. These problems and future work will specifically talk about problems with modelling functionality in Java. Other problems not relating specifically to Java functionality will be discussed in chapter 7.

Finally it should be noted that many of the API classes we model inherit from other API classes. In our examples we have ignored and removed the inherited classes in most cases. This is for two reasons.

1. To reduce the size of the examples as we want to focus on the specific modelling of the classes themselves, not the modelling of the super classes (except where we want to demonstrate specifically the modelling of a class with inheritance).

2. There are many methods in the inherited classes that we cannot model. This does not cause a problem with modelling inheritance itself but it does mean that as we can not fully model the inherited classes yet, we are unable in certain cases to model them when extended by another class.

6.1 Stack Example

Our first example is of a simple class that represents a Stack. In this case we will give the full example. First we will give the original Java Stack class along with the embedded equations for modelling the class algebraically.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 *
 * <hidden> op AStack : -> Stack .</hidden>
 */

public class Stack {

  /**
   *
   * <code>
   * eq Stack() = AStack .
   * </code>
   */
  public Stack() {
    ...
  }

  /**
   *
   * @return boolean
   * <code>
   * var S : Stack .
   * var O : Object .
   * eq (AStack).empty()q = true .
   * eq ((S).push(O)o).empty()q = false .
   * </code>
   */
  public boolean empty(){
    ...
  }

  /**
   *
   * @return Object
   * <code>
   * var S : Stack .
   * var O : Object .
   * eq ((S).push(O)o).peek()q = O .
 * </code>
 */
public Object peek(){
  ...
}

/**
 *
 * @return Object
 * <code>
 * var S : Stack .
 * var O : Object .
 * eq ((S).push(O)o).pop()q = O .
 * eq ((S).push(O)o).pop()o = S .
 * </code>
 */
public Object pop(){
  ...
}

/**
 *
 * @param o Object
 * @return Object
 */
public Object push(Object o){
  ...
}



/**
 *
 * @param o Object
 * @return int
 *
 * <code>
 * var S : Stack .
 * var O1 : Object .
 * var O2 : Object .
 * eq (AStack).search(O1)q = -1 .
```

```
   * eq ((S).push(O1)o).search(O1)q = 1 .
   * eq ((S).push(O2)o).search(O1)q = if ((O1 =/= O2) and
   *    ((S).search(O1)q == -1) ) then (-1)
   *    else (((S).search(O1)q) + 1) fi .
   * </code>
   *
   */
  public int search(Object o){
    ...
  }

}
```

We define an operator between the *hidden* tags called *AStack* to represent the base empty stack. This is used by the class' only constructor to create an empty stack instance. A stack is made up an empty stack *AStack* followed by a series of *push* calls which add an object to the top of the stack. As *push* is used as part of the structure of stack, we do not need to provide any reduction equations as we wish to retain the *push* calls. The *pop* command provides an example of a method that both returns a value and changes the state of a class instance. The method is defined by the following equations.

```
eq ((S).push(O)o).pop()q = O .
eq ((S).push(O)o).pop()o = S .
```

The first equation defines the query value *pop* by returning the object at the top of the stack, that is the object contained in the right outermost *push* call. The second equation defines the state change functionality of the *pop* method. It does this by removing the right outermost *push* command.

Another method of interest in this example is the *search* method. This returns the distance from the top of stack to the object being searched for. If the object being searched for is not in the stack then the method returns -1. At first this may appear to be easily defined as follows.

```
eq (AStack).search(O1)q = -1 .
eq ((S).push(O1)o).search(O1)q = 1 .
ceq ((S).push(O2)o).search(O1)q = (S).search(O1)q) + 1
    if O1 =/= O2 .
```

However if we are searching for an object that is not in a stack then this will return an incorrect result. For instance if the stack contained three objects then a search for an element not in the stack would return 2 when it should return -1. The correct way to define this method is as follows.

```
eq (AStack).search(O1)q = -1 .
eq ((S).push(O1)o).search(O1)q = 1 .
eq ((S).push(O2)o).search(O1)q = if ((O1 =/= O2) and
    ((S).search(O1)q == -1) ) then (-1) else
    (((S).search(O1)q) + 1) fi .
```

In this version we check the value returned by a recursive call to *search* to see if it has returned -1. If it does we retain the -1 value rather than add one to the recursive call's return value. This demonstrates how extra consideration needs to be used when creating equations for methods that need to keep track of the value returned by recursive calls.

The resulting ACS for the *Stack* class is as follows.

```
Class Stack Extends Object {
    Constructors{
        Stack :  .
    }
    Hidden{
        op AStack : -> Stack .
    }
    Methods{
        pop :  -> Object .
        push : Object -> Object .
        empty :  -> Bool .
        peek :  -> Object .
        search : Object -> Int .
    }
    Operations{
    }
    Variables{
        var O : Object .
        var S : Stack .
        var O1 : Object .
        var O2 : Object .
    }
    Equations{
        eq Stack() = AStack .
        eq (AStack).empty()q = true .
        eq ((S).push(O)o).empty()q = false .
        eq ((S).push(O)o).peek()q = O .
        eq ((S).push(O)o).pop()q = O .
        eq ((S).push(O)o).pop()o = S .
        eq (AStack).search(O1)q = -1 .
        eq ((S).push(O1)o).search(O1)q = 1 .
        eq ((S).push(O2)o).search(O1)q = if ((O1 =/= O2) and
            ((S).search(O1)q == -1) ) then (-1) else
            (((S).search(O1)q) + 1) fi .
    }
}
```

Finally we give the corresponding Maude code for modelling this class.

```
fmod STACK is

    protecting BUILDLINK .

    sort Stack .
    sort StackObject .
    sort StackBool .
    sort StackInt .
    sort StackArray .


    op Stack() :  -> Stack .

    op AStack : -> Stack .

    op _,_ : Stack Object -> StackObject .
    op oval(_) : StackObject -> Stack .
    op qval(_) : StackObject -> Object .
    op _,_ : Stack Bool -> StackBool .
    op oval(_) : StackBool -> Stack .
op qval(_) : StackBool -> Bool .
op _,_ : Stack Int -> StackInt .
op oval(_) : StackInt -> Stack .
op qval(_) : StackInt -> Int .
op _.pop() : Stack  -> StackObject .
op _.pop() : StackObject  -> StackObject .
op _.pop() : StackBool  -> StackObject .
op _.pop() : StackInt  -> StackObject .
op _.pop()o : Stack -> Stack .
op _.pop()q : Stack -> Object .
op _.push(_) : Stack  Object -> StackObject .
op _.push(_) : StackObject  Object -> StackObject .
op _.push(_) : StackBool  Object -> StackObject .
op _.push(_) : StackInt  Object -> StackObject .
op _.push(_)o : Stack Object -> Stack .
op _.push(_)q : Stack Object -> Object .
op _.empty() : Stack  -> StackBool .
op _.empty() : StackObject  -> StackBool .
op _.empty() : StackBool  -> StackBool .
op _.empty() : StackInt  -> StackBool .
op _.empty()o : Stack -> Stack .
op _.empty()q : Stack -> Bool .
op _.peek() : Stack  -> StackObject .
op _.peek() : StackObject  -> StackObject .
op _.peek() : StackBool  -> StackObject .
op _.peek() : StackInt  -> StackObject .
op _.peek()o : Stack -> Stack .
op _.peek()q : Stack -> Object .
op _.search(_) : Stack  Object -> StackInt .
op _.search(_) : StackObject  Object -> StackInt .
op _.search(_) : StackBool  Object -> StackInt .
op _.search(_) : StackInt  Object -> StackInt .
op _.search(_)o : Stack Object -> Stack .
op _.search(_)q : Stack Object -> Int .


op null : -> Stack .
op _[_] : StackArray Int -> Stack .
op _[_]:=_ : StackArray Int Stack -> StackArray .
op add : StackArray Int Stack -> StackArray .
op add : StackArray Stack -> StackArray .
op get : StackArray Int -> Stack .
op _.length : StackArray -> Int  .
op newStackArray : Int -> StackArray .
op new:Stack[_] : Int -> StackArray .
op EStackArray : -> StackArray .


var O : Object .
var S : Stack .
var O1 : Object .
var O2 : Object .
var SYS0 : Stack .
var SYS1 : Object .
var SYS2 : StackObject .
var SYS3 : Bool .
var SYS4 : StackBool .
var SYS5 : Int .
var SYS6 : StackInt .
var SYS7 : Object .
var SYS8 : Object .
var SYS9 : Stack .
var SYS10 : Stack .
var SYS11 : StackArray .
var SYS12 : Int .
var SYS13 : Int .

eq Stack() = AStack .
eq (AStack).empty()q = true .
eq ((S).push(O)o).empty()q = false .
eq ((S).push(O)o).peek()q = O .
eq ((S).push(O)o).pop()q = O .
eq ((S).push(O)o).pop()o = S .
eq (AStack).search(O1)q = -1 .
eq ((S).push(O1)o).search(O1)q = 1 .
eq ((S).push(O2)o).search(O1)q = if ((O1 =/= O2) and
    ((S).search(O1)q == -1) ) then (-1) else
    (((S).search(O1)q) + 1) fi .
eq oval(SYS0,SYS1) = SYS0 .
eq qval(SYS0,SYS1) = SYS1 .
eq oval(SYS0,SYS3) = SYS0 .
eq qval(SYS0,SYS3) = SYS3 .
eq oval(SYS0,SYS5) = SYS0 .
eq qval(SYS0,SYS5) = SYS5 .
eq (SYS0).pop() = (SYS0).pop()o,(SYS0).pop()q .
eq (SYS2).pop() = (oval(SYS2)).pop()o,
   (oval(SYS2)).pop()q .
eq (SYS4).pop() = (oval(SYS4)).pop()o,
   (oval(SYS4)).pop()q .
eq (SYS6).pop() = (oval(SYS6)).pop()o,
   (oval(SYS6)).pop()q .
eq (SYS0).push(SYS7) = (SYS0).push(SYS7)o,
   (SYS0).push(SYS7)q .
eq (SYS2).push(SYS7) =
   (oval(SYS2)).push(SYS7)o,(oval(SYS2)).push(SYS7)q .
eq (SYS4).push(SYS7) =
   (oval(SYS4)).push(SYS7)o,(oval(SYS4)).push(SYS7)q .
eq (SYS6).push(SYS7) =
   (oval(SYS6)).push(SYS7)o,(oval(SYS6)).push(SYS7)q .
eq (SYS0).empty() = (SYS0).empty()o,(SYS0).empty()q .
eq (SYS2).empty() = (oval(SYS2)).empty()o,
   (oval(SYS2)).empty()q .
eq (SYS4).empty() = (oval(SYS4)).empty()o,
   (oval(SYS4)).empty()q .
eq (SYS6).empty() = (oval(SYS6)).empty()o,
   (oval(SYS6)).empty()q .
eq (SYS0).peek() = (SYS0).peek()o,(SYS0).peek()q .
eq (SYS2).peek() = (oval(SYS2)).peek()o,
   (oval(SYS2)).peek()q .
eq (SYS4).peek() = (oval(SYS4)).peek()o,
   (oval(SYS4)).peek()q .
eq (SYS6).peek() = (oval(SYS6)).peek()o,
   (oval(SYS6)).peek()q .
eq (SYS0).search(SYS8) = (SYS0).search(SYS8)o,
   (SYS0).search(SYS8)q .
eq (SYS2).search(SYS8) =
   (oval(SYS2)).search(SYS8)o,(oval(SYS2)).search(SYS8)q .
eq (SYS4).search(SYS8) =
   (oval(SYS4)).search(SYS8)o,(oval(SYS4)).search(SYS8)q .
eq (SYS6).search(SYS8) =
   (oval(SYS6)).search(SYS8)o,(oval(SYS6)).search(SYS8)q .
```

```
    ceq get(add(SYS11,SYS9),SYS12) = SYS9
        if SYS12 = (SYS11).length .
    ceq get(add(SYS11,SYS9),SYS12) = get(SYS11,SYS12)
        if SYS12 =/= (SYS11).length .
    eq (EStackArray).length = 0 .
    eq (add(SYS11,SYS9)).length = (SYS11).length + 1 .
    ceq add(add(SYS11,SYS10),SYS12,SYS9) =
        add(SYS11,SYS9) if SYS12 = (SYS11).length .
    ceq add(add(SYS11,SYS10),SYS12,SYS9) =
        add(add(SYS11,SYS12,SYS9),SYS10) if SYS12 <=
        (SYS11).length .
    eq newStackArray(0) = EStackArray .
    eq newStackArray(SYS12) = add(newStackArray(SYS12 - 1),null) .
    eq (SYS11)[(SYS12)] = get(SYS11,SYS12) .
    eq ((SYS11)[(SYS12)]:=(SYS9)) = add(SYS11,SYS12,SYS9) .
    eq new:Stack[SYS12] = newStackArray(SYS12) .


endfm
```

It should be noted that the last set of equations define the array type and operations for the *Stack* class. The *Stack* class provided no functionality that we can not model. Hence it serves as a good example of the capabilities of our model.

6.2 ArrayList Example

For this and the rest of the examples we will not give the full example class and corresponding specification code, but will instead focus on sections of the example which show interesting modelling concepts and problems. An *ArrayList* structure is defined in a similar way to the *Stack* example. An *ArrayList* is defined as an empty *ArrayList* followed by *add* calls which add objects to the *ArrayList*. There are several methods which are of interest. The first we will examine is the *set* method that adds an object at a specified index in the *ArrayList*.

```
 /**
   *
   * @param i
   * @param x
   * @return
   *
   * <code>
   * var A : ArrayList .
   * var I : Int .
   * vars X Y : Element .
   * ceq ((A).add(Y)o).set(I,X)o = (A).add(X)o
   *    if I = (A).size()q .
   * ceq ((A).add(Y)o).set(I,X)o = ((A).set(I,X)o).add(Y)o
   *    if I =/= (A).size()q .
   * eq (A).set(I,X)q = (A).get(I)q .
   * </code>
   */
  public Element set(int i, Element x) {
    ...
  }
```

The problem with this method is that it uses an index to place the element to be added into the *ArrayList*. However the structure of *ArrayList* has no indexing component. In the original code, the indexing is most likely incorporated through private methods and fields. However our model is not interested in the private or protected components of a class only the public interface so we need to incorporate the indexing using only the public methods available. We do this using the *size* method. Let us look at the equations that define the state change component of the method *set*.

```
ceq ((A).add(Y)o).set(I,X)o = (A).add(X)o if I = (A).size()q .
ceq ((A).add(Y)o).set(I,X)o = ((A).set(I,X)o).add(Y)o
    if I =/= (A).size()q .
```

The first equation says that if the index position we are looking for is equal to the value returned by *size* when called on the *ArrayList* minus the current right outermost element then we add the object at the current position and discard the old object at that position. We add it using the *add* method which is used in the structure of *ArrayList*. The second equation recursively calls the set method on the *ArrayList* if the index value is not equal to the size of the *ArrayList* minus the right outermost element. We attach the right outermost element to the result of the recursive call of *set* on the rest of *ArrayList.*

This method also has a query return value so we need to provide an equation to define this.

```
eq (A).set(I,X)q = (A).get(I)q .
```

The query return value is the object currently held at index *I* (that is the object at position *I* before *set* replaces it with the new object). The above equation does this quite simply by calling the *get* method and passing to it the integer index *I.*

The *get* method is defined by the following equations.

```
ceq ((A).add(X)o).get(I)q = X if I = (A).size()q .
ceq ((A).add(X)o).get(I)q = (A).get(I)q if I =/= (A).size()q .
```

This works in a similar way to the *set* method but when it finds the correct index position it returns the object at that position. As can be seen the *set* method is able to use the definition of *get* to define part of its functionality thus reducing the need for repeated equations. As stated all of these methods make use of the *size* method. We will now look at its definition.

```
/**
   *
   * @return
   * <code>
   * var A : ArrayList .
   * var X : Element .
   * eq (Elist).size()q = 0 .
   * eq ((A).add(X)o).size()q = (A).size()q + 1 .
   * </code>
   */
  public int size() {
    ...
  }
```

We define *size* using the following two equations.

```
eq (Elist).size()q = 0 .
eq ((A).add(X)o).size()q = (A).size()q + 1 .
```

The first equation defines the size of an empty *ArrayList* as zero. The second equation recursively calls *size* on *ArrayList* when it is non-empty and adds one to the result. This means that in no way do the elements of an *ArrayList* have an actual index assigned to them. Their index positions are defined by the *size* method. Therefore if a new element is inserted into an *ArrayList* the index of all the subsequent elements will increase by one. This is the expected functionality of *ArrayList* as specified by the API documentation [Sun05g].

Next we will examine two methods that are used to obtain the index locations of a given element of an *ArrayList*.

```
/**
   *
   * @param x
   * @return
   *
   * <code>
   * var A : ArrayList .
   * vars X Y : Element .
   * eq (Elist).lastIndexOf(X)q = -1 .
   * eq ((A).add(X)o).lastIndexOf(X)q = (A).size()q .
   * ceq ((A).add(Y)o).lastIndexOf(X)q =
   *    (A).lastIndexOf(X)q if Y =/= X .
   * </code>
   */
  public int lastIndexOf(Element x) {
    ...
  }
```

This function returns the last index of a given element in the *ArrayList* (that is the right outermost element in the *ArrayList* structure). We define the function with the following three equations.

```
eq (Elist).lastIndexOf(X)q = -1 .
eq ((A).add(X)o).lastIndexOf(X)q = (A).size()q .
ceq ((A).add(Y)o).lastIndexOf(X)q =
    (A).lastIndexOf(X)q if Y =/= X .
```

The first equation returns -1 if the *ArrayList* is empty. That is, we have searched through the entire *ArrayList* and have not found an entry for the given object. The second equation returns the array position of the object we are searching for if we have found it. The equation does this by returning the current size of the *ArrayList* minus its right outermost element, if that element is the object we are looking for. That is it returns the index position of that element according to the definition of *lastIndexOf* (note that if you have an *ArrayList* of size *N* then the *ArrayList* indexes from *0* to *N-1*). The third equation recursively calls *lastIndexOf* on the remainder of an *ArrayList* if the right outermost element is not equal to the object we are searching for.

Next we will look at the *indexOf* method.

```
/**
 *
 * @param x
 * @return
 *
 * <code>
 * var A : ArrayList .
 * vars X Y : Element .
 * eq (Elist).indexOf(X)q = -1 .
 * ceq (A).indexOf(X)q = (A).lastIndexOf(X)q
 *    if ((((A).remove((A).lastIndexOf(X)q)o).contains(X)q)
 *    == false) .
 * eq ((A).add(Y)o).indexOf(X)q = (A).indexOf(X)q .
 * </code>
 */
public int indexOf(Element x) {
  ...
}
```

Note that the *vars* keyword is used in Maude as a way of declaring multiple variables of the same type. We use it here simply as a notational convenience.

This method returns the index of the first occurrence of a given object in an *ArrayList* (that is the left innermost occurrence of the object). This is more complicated than the *lastIndexOf* method. We cannot easily keep searching back through the *ArrayList* to find the first occurrence of the object as we will not know when to stop once we have found it. The answer to this is to remove the later repetitions of the object from the *ArrayList* until we are left with the only the first occurrence of the object in the *ArrayList*. Then all we have to is to call *lastIndexOf* which will now be equivalent to *indexOf*.

We do this using the following equations.

```
eq (Elist).indexOf(X)q = -1 .
ceq (A).indexOf(X)q = (A).lastIndexOf(X)q
    if ((((A).remove((A).lastIndexOf(X)q)o).contains(X)q)
    == false) .
eq ((A).add(Y)o).indexOf(X)q = (A).indexOf(X)q .
```

The first equation returns *-1* on empty *ArrayList* (that is the *ArrayList* does not contain an occurrence of the object we are searching for). The second equation returns the *lastIndexOf* of the element we are searching for. It is only used if when we find the *lastIndexOf* of the object we are searching for, then remove that element and the new *ArrayList* does not contain any more occurrences of that object. That is it is called if the *ArrayList* contains only one occurrence of the object we are searching for. To do this it uses two of *ArrayList*'s other methods. These methods are *remove* which removes an object at a given index and *contains* which returns *true* if an *ArrayList* contains a given object and *false* if it does not. The third equation recursively calls *indexOf* on the *ArrayList* minus its right outermost element. It is only called if the first two equations fail.

Finally we will look at the *removeRange* method.

```
  /**
   *
   * @param i
   * @param j
   *
   * <code>
   * var A : ArrayList .
   * vars I J : Int .
   * eq (A).removeRange(I, I)o = A .
   * eq (A).removeRange(I, J)o =
   *    ((A).remove(I)o).removeRange(I, J - 1)o .
   * </code>
   */
  public void removeRange(int i, int j) {
    ...
  }
```

This method removes all the elements between the range of *I* - *J* where *I* is inclusive and *J* is not.

```
eq (A).removeRange(I, I)o = A .
eq (A).removeRange(I, J)o =
    ((A).remove(I)o).removeRange(I, J - 1)o .
```

The first equation returns the array unchanged if the range start point is equal to the range end point. This ensures that the *J*'th position of the range is never removed (keeping it exclusive). The second equation removes the element at the start of the range and then recursively calls *removeRange* on the result with the end range value *J* reduced by *1*.

A large percentage of the *ArrayList* class functionality has been modelled. There are however a few components of this class that we are at present unable to model. We will now look at each of these components.

The first problem are two of *ArrayList*'s constructors. The first allows you to pass in a class instance of the *Collection* class to initialise the *ArrayList* with the elements in the collection. *Collection* is a Java Interface class which means it purely defines syntax but no semantics. At present was are unable to model Java Interfaces so we cannot model this method. However we feel that this could be incorporated through the correct use of Maude subsorts and membership axioms. In Java, sub classes of *Collection* that fully define the semantics are passed as class instances to this constructor. We feel the same could be done using subsorting in Java. The second constructor allows the user to pass in an integer to ensure the initialised *ArrayList* is of a given capacity. This is essentially to help make code more efficient. However in our model we do not model the underlying memory or elements usually hidden from the user. Although we might be able to set the internal capacity it would make no difference to the structure of the *ArrayList*. At present we see no way of modelling this sort of functionality without completely altering the model to accommodate the hidden underlying system.

For similar reasons we cannot model the *addAll*, *ensureCapacity*, and *trimToSize* as they all involve either the internal capacity or use of the *Collection* interface.

6.3 Label Example

Our next example looks at how our specification techniques can be used to model parts of the Java GUI component classes. The class we have chosen to model is the *Label* class. It was while modelling this class that we altered our modelling technique to accommodate multiple methods that can define the structure (such as the field values) of a class instance. This new modelling technique is at present experimental and has only been used in this one case as an example solution to the problem of classes with multiple definitions of its structure. This technique requires further investigation and refinement before it is fully implemented into our model. All our previous examples have only had one method forming part of the class structure. However *Label* has two. In order to accommodate this we have to declare which methods they are and what is the type of the value that they set in the structure.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 *
 * <hidden> op ALabel : -> Label .</hidden>
 * <struct>
 * setText
 * String
 * setAlignment
 * Int
 * </struct>
 */
```

```
public class Label {
```

These extra declarations are placed between the *struct* keywords.

```
<struct>
setText
String
setAlignment
Int
</struct>
```

This declares that the structure of *Label* can consist of a method called *setText* which sets a String value and a method called *setAlignment* that sets an integer value. We will look at two methods that will need to make use of *setText* and *setAlignment.* The first method is *getText.*

```
  /**
   *
   * @return String
   * <code>
   * eq (L).getText()q = (L).setText[...] .
   * </code>
   */
  public String getText(){
    ...
  }
```

The method is defined with a single equation.

```
eq (L).getText()q = (L).setText[...] .
```

This uses the operator *setText[...]* which can be generated from the information between the *struct* statements. *setText[...]* is used to obtain the current value set by *setText[...].* The equations for *setText[...]* are given below.

```
eq ((SYSO).setAlignment(SYS7)o).setText[...] =
    (SYSO).setText[...] .
eq ((SYSO).setText(SYS8)o).setText[...] = SYS8 .
```

This searches through a *Label*'s structure ignoring any *setAlignment* calls until it finds a *setText* call. When it finds a *setText* call it returns the string value parameter of *setText*.

The second method we will look at, *getAlignment*, is defined in a similar way.

```
eq (L).getAlignment()q = (L).setAlignment[...] .
```

The equations that define *setAlignment[...]* are similar to those defining *setText[...]*.

```
eq ((SYSO).setAlignment(SYS7)o).setAlignment[...] = SYS7 .
eq ((SYSO).setText(SYS8)o).setAlignment[...] =
    (SYSO).setAlignment[...] .
```

We do however have a problem with the modelling of *getAlignment* and the *setAlignment*. In the original Java code you can set the alignment value by passing in one of the class' constant integers *LEFT*, *CENTER*, and *RIGHT*. The actual values of these are hidden from the user. However it is possible that in some cases when this way of passing integer values is used, it will be necessary to model the actual values as well. At present our model doesn't do this. Although it wouldn't be hard to do, it would still mean that a knowledge of the underlying Java code of a class would be needed in order to discover the actual values of these constants.

There are two methods that we are currently unable to model in the *Label* class. The first *addNotify* is used to control how the label appears on screen. We are currently unable to model the on screen behaviour of the GUI functionality of Java classes. There are possible ways to model this sort of functionality. One suggestion is to have the Maude specification generate a bitmap that will represent the current state of the on screen display. However although this is possible it is probably not desirable. The type of modelling we use is not interested in modelling GUI classes, but in modelling computational classes, and storage classes. We only mention the GUI aspect to emphasize that although we do not think it is neccessary to model it, we believe it is possible to do so.

The other method we cannot model is *getAccessibleContext.* This returns an *AccesibleContext* instance which contains more information on the *Label* instance. We are unable to model this method because we have not modelled *AccessibleContext. AccessibleContext* is tied heavily in with GUI aspects of Java and thus it is at present not possible to model this class.

6.4 Geometric Examples

The following set of examples are largely taken from the *java.awt.geom* package which contains a collection of geometric themed classes, such as *Line* and *Rectangle.* However we have adapted the classes contained in this package as the majority of them are abstract. We have used classes contained both in the *java.awt* and *java.awt.geom* that expand the classes we have modelled to complete the abstract methods. We are unable at present to model abstract classes so we have chosen to adapt them using these extended classes. The original Java classes also use floating point numbers for the coordinate system. Our model at present only models integer numbers so for the purpose of this example, the classes have been changed to use integer numbers. This does generate a few issues when modelling certain aspects of these classes and we will discuss those later. The aim of modelling these classes is to demonstrate a good example of modelling classes that provide a high level of computation functionality.

6.4.1 Point2D Example

This example is used to model a point in a cartesian coordinate system. In Java the origin (0,0) is in the top left corner of the coordinate system, but this does not affect the calculations and would only be noticeable if displayed on a screen. This applies to all the geometric examples.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 *
 * <hidden>op APoint2D : -> Point2D .</hidden>
 */

public class Point2D {

  public int x;
  public int y;


  /**
   *
   * @param x int
   * @param y int
   * <code>
   * var X : Int .
   * var Y : Int .
   * eq Point2D(X,Y) = ((APoint2D).x:=(X)).y:=(Y) .
   * </code>
   */
  public Point2D(int x,int y){
    ...
  }
```

Structurally a *Point2D* is defined simply by its *x,y* coordinates which are stored in fields *x* and *y*. There are several constructors that can be used to create a *Point2D* instance. These are all variations on setting the *x,y* coordinates by passing in these values in different ways such as another *Point2D* instance or not passing in anything in which case the *x,y* coordinates are both set to zero.

Next we will look at the *clone1* and *equals1* methods.

```
  /**
   *
   * @return Object
   * <code>
  * var P : Point2D .
  * eq (P).clone1()q = Point2D(((P).x),((P).y)) .
  * </code>
  */
public Point2D clone1(){
  ...
}

/**
  *
  * @param O Object
  * @return boolean
  * <code>
  * var P : Point2D .
  * var OB : Point2D .
  * ceq (P).equals1(OB)q = true if (((P).x == (OB).x) and
  *    ((P).y == (OB).y)) .
  * </code>
  */
public boolean equals1(Point2D O){
  ...
}
```

The functionality of these methods is relatively simple. The method *equals1* takes in another point and checks to see if the two points are equal by checking to see if they have the same x,y coordinates. The *clone1* method creates a new *Point2D* with the current *Point2D*'s x,y coordinates. These are meant to be an implementation of *Point2D's clone* and *equals* methods which override methods of the same name inherited from *Object.* The *equals* method should take in a general *Object* instance and *clone* should return one. In actuality these will always be *Point2D* instances which are subtypes of *Object* (if you tried to pass in an instance that was not of the subclass type *Point2D* to the *equals* method you would generate an exception which at present we can not model). In order to model this exactly we would need to model class casting which at present we don't. It might be possible to do this by using Maude membership axioms together with Maude subsorts. In order for us to model the functionality we have therefore changed the *Object* inputs and return types to *Point2D.* However as *Point2D* is a subtype of *Object* Java does not allow you to name these new function *equals* and *clone* as this will cause a signature clash (it will not override the methods as *Point2D* is a subtype of *Object* not a completely separate type) so we have renamed our methods *equals1* and *clone1* for the purpose of this example.

Next we will look at the *distance* method.

```
/**
  *
  * @param i int
  * @return int
  * <code>
  * var P : Point2D .
  * var X1 : Int .
  * var X2 : Int .
  * var Y1 : Int .
  * var Y2 : Int .
  * eq (P).distance(X1,X2,Y1,Y2)q = isqrt(sqr(X2 - X1) +
  *     sqr(Y2 - Y1)) .
  * </code>
  */
 public int distance(int x1,int x2,int y1, int y2){
   ...
 }
```

This method calculate the distance between two points. The distance is the square root of the sum of the squares of the difference between the x coordinates and the y coordinates $D = \sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}$. However this poses a problem for us when modelling in Maude. Maude only has a square root function that works with floating numbers and we are using integer numbers. We therefore define a function in the built-in Maude modules that allows us to find the square root of an integer number.

```
op isqrt : Int -> Int .

eq isqrt(I) = rat(floor(sqrt(float(I)))) .
```

The above equations make use of Maude's conversion module. It first converts the integer that you wish to find the square root of to a floating point number. It then uses Maude's floating point square root function to find the square root of this number. Next it uses Maude's *floor* function to find the floor of the square root. Finally it uses Maude's *rat* function to convert the floating number back to an integer. As can be seen one of the disadvantages of using integer numbers are rounding errors in this case resulting from taking the floor of the floating point square root (although it should be noted that even floating point numbers will suffer from rounding errors albeit to a less degree than integers).

We also for convenience have defined a function to square integer numbers.

```
op sqr : Int  -> Int .

eq sqr(I) = I * I .
```

The next method we will look at is *distanceSq*

```
/**
 *
 * @param x1 int
 * @param x2 int
 * @param y1 int
 * @param y2 int
 * @return int
 *
 * <code>
 * var X1 : Int .
 * var X2 : Int .
 * var Y1 : Int .
 * var Y2 : Int .
 * eq (P).distanceSq(X1,X2,Y1,Y2)q =
 *     sqr((P).distance(X1,X2,Y1,Y2)q) .
 * </code>
 */
public int distanceSq(int x1,int x2,int y1,int y2){
  return 0;
}
```

This method returns the square of the distance. We have coded this method so that it squares the result of a call to the *distance* method. However this is not actually the correct functionality. In actual fact this should do exactly what the *distance* method does but it should not take the square root. If the square root is taken and then squared back again then there is the danger of rounding errors as even with floating point numbers the square root function cannot return an exact value in some cases. This demonstrates the importance of formal documentation. The informal documentation that is provided with this class does not make the importance of this function clear and thus there is the danger of misinterpreting its functionality as we have done deliberately to emphasize a point. The correct way to model the functionality of this method is to have it calculate the distance itself and not take the square root. The *distance* method could then make use of this function by calling it and finding the square root of the result.

We have not been able to model *hashCode* which returns the hashcode for this instance. As this is part of the internal working of the Java class we are unable to determine how the hashcode is generated easily. Also we are unable to model *toString.* This returns a string representation of the class instance but as the representation can vary between platforms it is not possible to accurately model this method. It would be possible to model it in a general way but it would not be a completely accurate model of the method's functionality.

6.4.2 Line2D Example

The next geometric example class we will look at is *Line2D.* This class is used to model a line in a cartesian coordinate system.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 *
 * <hidden>op ALine2D : -> Line2D .</hidden>
 */

public class Line2D {
  public int x1;
  public int x2;
  public int y1;
  public int y2;

  /**
   *
   * @param x1 int
   * @param y1 int
   * @param x2 int
   * @param y2 int
   * <code>
   * var X1 : Int .
   * var X2 : Int .
   * var Y1 : Int .
   * var Y2 : Int .
   * eq Line2D(X1,Y1,X2,Y2) =
   *    ((((ALine2D).x1:=(X1)).y1:=(Y1)).x2:=(X2)).y2:=(Y2) .
   * </code>
   */
  public Line2D(int x1,int y1,int x2,int y2){
    ...
  }
```

Structurally a *Line2D* is defined as two sets of *x,y* coordinates which are stored in fields *x1*, *y1*, *x2*, and *y2*. The constructor above allows a user to create a new *Line2D* instance by passing in values for these fields. Similar to *Point2D* there are other constructors that allow us to create a *Line2D* instance by passing in values for the fields in different formats, such as another *Line2D* or two *Point2D* instances.

We will now look at some of the class' methods.

```
  /**
   *
   * @return Point2D
   * <code>
   * var L : Line2D .
   * eq (L).getP1()q = Point2D((L).x1,(L).y1) .
   * </code>
   */
  public Point2D getP1(){
    ...
  }
```

The above method retrieves the first point of a *Line2D* instance. It does this by creating a new *Point2D* instance using the *Line2D*'s *x1* and *y1* coordinates. A similar method exists for the *Line2D*'s second point. In order to do this the algebraic specification for *Point2D* needs to be created first. It then needs to be loaded into Maude first and then *Line2D* needs to protect the *Point2D* module and be loaded in itself. This however would present a problem if *Point2D* makes use of *Line2D* as we would have a cyclical dependency and Maude would not be able to load in the modules due to missing operators. One solution to this would be to create one large Maude module with all the classes to be modelled in it. However this causes a problem for the automated program conversion program as it could result in name clashes with variables. A better solution is to remove all the equations from each class to be modelled and place them into a separate Maude module to be loaded last. This will allow all operators to be declared before they are defined in the equations as being dependent on each other. So although in this case there are no cyclical dependencies it is important to recognise this problem and its potential solutions as there are cases in which it could occur.

Next we will look at one of *Line2D*'s multiple *contains* methods.

```
  /**
   *
   * @param p Point2D
   * @return boolean
   * <code>
   * eq (L).contains(P1)q = false .
   * </code>
   */
  public boolean contains(Point2D p){
    ...
  }
```

The above method checks to see if the *Line2D* instance contains a *Point2D* instance. However a *Line2D* is a one dimensional object and thus can not contain anything. Hence this method and all the other *contains* methods return *false.*

The next method we will look at is *relativeCCW.*

```
/**
 *
 * @param x1 int
 * @param y1 int
 * @param x2 int
 * @param y2 int
 * @param px int
 * @param py int
 * @return int
 * <code>
 * var PX : Int .
 * var PY : Int .
 * ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = -1 if
 *    ((((PX - X1) * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1)))
 *    == 0) and ((((PX - X1)  * (Y2 - Y1)) + ((PY - Y1) *
 *    (X2 - X1))) < 0) .
 * ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 1 if
 *    ((((PX - X1) * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1)))
 *    == 0) and ((((PX - X1)  * (Y2 - Y1)) + ((PY - Y1) *
 *    (X2 - X1))) > 0) .
 * ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 0 if
 *    ((((PX - X1) * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1)))
 *    == 0) and (((PX - X1)  * (Y2 - Y1)) + ((PY - Y1) *
 *    (X2 - X1))) == 0) .
 * ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 0 if
 *    ((((PX - X1) * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1)))
 *    > 0) and (((((PX - X1) - X2)  * (Y2 - Y1)) +
 *    (((PY - Y1) - Y2) * (X2 - X1))) <= 0) .
 * ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = 1 if
 *    ((((PX - X1) * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1)))
 *    > 0) and (((((PX - X1) - X2)  * (Y2 - Y1)) +
 *    (((PY - Y1) - Y2) * (X2 - X1))) > 0) .
 * ceq (L).relativeCCW(X1,Y1,X2,Y2,PX,PY)q = -1 if
 *    ((((PX - X1) * (Y2 - Y1)) - ((PY - Y1) * (X2 - X1)))
 *    < 0) .
 * </code>
```

```
  *
  */
 public int relativeCCW(int x1,int y1,int x2,int y2,int px,int py){
   ...
 }
```

The above method is used to determine where a given point lies in relation to a line. In order to be able to create equations for the function it was neccessary to look at how the source code works and try and create equations that do the same thing. This demonstrates the importance of formal modelling being done at the design phase. As we had to resort to mimicking the source code, any errors in the source code will have been duplicated in the equations. Also the source code is at times hard to interpret and therefore there is the danger of new errors being introduced as we try and determine the source code's original functionality. There is also a lot of repetition of calculations due to the differences in how equations work and how the original source code works. The above shows how it is important the programmers, designers, and formal modelers should work closely together during every phase of the creation of new code. There are several methods in this example and others where we have had to resort to trying to mimic the original source code.

Finally we will look at the *linesIntersect* method.

```
/**
  *
  * @param x1 int
  * @param y1 int
  * @param x2 int
  * @param y2 int
  * @param x3 int
  * @param y3 int
  * @param x4 int
  * @param y4 int
  * @return boolean
  * <code>
  * var X3 : Int .
  * var Y3 : Int .
  * var X4 : Int .
  * var Y4 : Int .
  * ceq (L).linesIntersect(X1,Y1,X2,Y2,X3,Y3,X4,Y4)q = true
  *    if ((((L).relativeCCW(X1, Y1, X2, Y2, X3, Y3)q) *
  *    ((L).relativeCCW(X1, Y1, X2, Y2, X4, Y4)q)) <= 0)
  *    and ((((L).relativeCCW(X3, Y3, X4, Y4, X1, Y1)q) *
  *    ((L).relativeCCW(X3, Y3, X4, Y4, X2, Y2)q)) <= 0))
  *    == true .
  * eq (L).linesIntersect(X1,Y1,X2,Y2,X3,Y3,X4,Y4)q =
  *    false .
  * </code>
  */
public boolean linesIntersect(int x1,int y1,int x2,
              int y2,int x3,int y3,int x4,int y4){
  ...
}
```

The above method checks to see if one line intersects another. It does this using *relativeCCW* to check to see where each point of the one line lies in relation to the other. There are two potential problems with this method. The first relates to the fact that we are using integers and not floating point numbers for the coordinates of the lines. This could potentially lead to rounding errors erroneously reporting that a line intersects the other line when it does not or vice versa. This problem can be improved by adapting the model to incorporate floating numbers. The other problem relates to *relativeCCW*. Although it is not hard to see how we can use *RelativeCCW* in this method to check if a line intersects another line we run the risk of the possible bugs in the equations for *relativeCCW* as mentioned earlier causing inaccurate results for *linesIntersect*. This shows that errors generated earlier on in a different method can effect later methods that use the potentially flawed method.

There are two methods both called *getPathIterator* which returns an iteration instance that defines the boundary of the *Line2D* object. We have not modelled these methods for two reasons. The first is that it returns a Java interface instance and as already stated our model at present does not model interfaces. Also it takes in an *AffineTransform* class instance as an input which, although we are able to model, we have not modelled this class due to time limitations.

6.4.3 Dimension2D Example

This class is used to model a dimension for use with our *Rectangle2D* example that we will look at later.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 * <hidden>op ADimension2D : -> Dimension2D .</hidden>
 */
public class Dimension2D {

  public int height;
  public int width;


  /**
   *
   * @param w int
   * @param h int
   * <code>
   * var W : Int .
   * var H : Int .
   * eq Dimension2D(W,H) = (ADimension2D).setSize(W,H)o .
   * </code>
   */
  public Dimension2D(int w,int h){
    ...
  }
```

A *Dimension2D* is defined as a width and a height dimension which are stored in integer fields *width* and *height.* The constructor above allows a user to create a *Dimension2D* by passing in values for these fields. Like in the previous examples other constructors exist for this class.

We will not examine any of the methods here. They are relatively simple and do not have any interesting features. The only methods we could not model are *hashCode* and *toString* for the same reasons we gave in the other geometric examples. The purpose of modelling this class is to allow us to more fully model the functionality of the *Rectangle2D* as some of *Rectangle2D*'s methods and constructors make use of *Dimension2D*

6.4.4 Rectangle2D Example

The final class we will look at in our geometric set of examples is *Rectangle2D*. This class is used to model rectangles in cartesian coordinate systems.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 *
 * <hidden>op ARectangle2D : -> Rectangle2D .</hidden>
 */
public class Rectangle2D {

  public int height;
  public int width;
  public int x;
  public int y;

  /**
   *
   * @param x int
   * @param y int
   * @param width int
   * @param height int
   * <code>
   * var X : Int .
   * var Y : Int .
   * eq Rectangle2D(X,Y,W,H) =
   *    ((((ARectangle2D).x:=(X)).y:=(Y)).width:=(
   *        W)).height:=(H) .
   * </code>
   */
  public Rectangle2D(int x,int y,int width,int height){
    ...
  }
```

Structurally a *Rectangle2D* is defined as a set of *x,y* coordinates representing the top left corner of the *rectangle* and a width and a height. These are stored in integer fields *x*, *y*, *width*, and *height* respectively. The constructor shown above allows a user to create a new *Rectangle2D* by passing in values for these fields. As in the other geometric examples there are other constructors that allow you to pass in values for these fields in a different way, such as another *Rectangle2D* instance or a *Point2D* and *Dimension2D* instance.

First we will look at the *add* method.

```
/**
   *
   * @param x int
   * @param y int
   * <code>
   * var R : Rectangle2D .
   * var X : Int .
   * var Y : Int .
   * ceq (R).add(X,Y)o = ((((R).width:=(((R).x + (R).width) -
   *    X)).x:=(X)).height:=(((R).y + (R).height) - Y)
   *    ).y:=(Y) if (X < (R).x) and (Y < (R).y) .
   * ceq (R).add(X,Y)o = (((R).width:=(((R).x + (R).width) -
   *    X)).x:=(X)).height:=(Y - (R).y) if (X < (R).x) and
   *    (Y > ((R).y + (R).height)) .
   * ceq (R).add(X,Y)o = (R).width:=(X - (R).x)  if
   *    (X > ((R).x + (R).width)) and (Y >= (R).y) and (Y
   *    <= ((R).y + (R).height)) .
   * ceq (R).add(X,Y)o = (((R).width:=(X - (R).x)).height:=
   *    (((R).y + (R).height) - Y)).y:=(Y) if (X > ((R).x +
   *    (R).width)) and (Y < (R).y) .
   * ceq (R).add(X,Y)o = ((R).width:=(X - (R).x)).height:=(Y -
   *    (R).y) if (X > ((R).x + (R).width)) and (Y > ((R).y +
   *    (R).height)) .
   * ceq (R).add(X,Y)o = R if (X >= (R).x) and (X <= ((R).x +
   *    (R).width)) and (Y >= (R).y) and (Y <= ((R).y +
   *    (R).height)) .
   * ceq (R).add(X,Y)o = ((R).height:=(((R).y + (R).height) -
   *    Y)).y:=(Y) if (X >= (R).x) and (X <= ((R).x +
   *    (R).width)) and (Y < (R).y) .
   * ceq (R).add(X,Y)o = (R).height:=(Y - (R).y) if (X >=
   *    (R).x) and (X <= ((R).x + (R).width)) and (Y > ((R).y
   *    + (R).height)) .
   * ceq (R).add(X,Y)o = R [owise] .
   </code>
   */
  public void add(int x,int y){
    ...
  }
```

This method allows you to pass in an *x,y* coordinate. The method then changes the *Rectangle2D* instance so that it contains the original rectangle and the *x,y* coordinate in the smallest possible size. This is done with the following equations.

```
ceq (R).add(X,Y)o = ((((R).width:=(((R).x + (R).width) - X)
    ).x:=(X)).height:=(((R).y + (R).height) - Y)).y:=(Y)
    if (X < (R).x) and (Y < (R).y) .
ceq (R).add(X,Y)o = (((R).width:=(((R).x + (R).width) -
    X)).x:=(X)).height:=(Y - (R).y) if (X < (R).x) and (Y >
    ((R).y + (R).height)) .
ceq (R).add(X,Y)o = (R).width:=(X - (R).x)  if (X > ((R).x +
    (R).width)) and (Y >= (R).y) and (Y <= ((R).y +
    (R).height)) .
ceq (R).add(X,Y)o = (((R).width:=(X - (R).x)).height:=
    (((R).y + (R).height) - Y)).y:=(Y) if (X > ((R).x +
    (R).width)) and (Y < (R).y) .
ceq (R).add(X,Y)o = ((R).width:=(X - (R).x)).height:=(Y -
    (R).y) if (X > ((R).x + (R).width)) and (Y > ((R).y +
    (R).height)) .
ceq (R).add(X,Y)o = R if (X >= (R).x) and (X <= ((R).x +
    (R).width)) and (Y >= (R).y) and (Y <= ((R).y +
    (R).height)) .
ceq (R).add(X,Y)o = ((R).height:=(((R).y + (R).height) -
    Y)).y:=(Y) if (X >= (R).x) and (X <= ((R).x + (R).width))
    and (Y < (R).y) .
ceq (R).add(X,Y)o = (R).height:=(Y - (R).y) if (X >= (R).x)
    and (X <= ((R).x + (R).width)) and (Y > ((R).y +
    (R).height)) .
ceq (R).add(X,Y)o = R [owise] .
```

There are nine possible different scenarios that could occur when we add the point. They are that the point is either directly left, directly right, directly up, directly down, up and to the left, up and to the right, down and to the left, down and to the right, and inside the rectangle. This is reflected in the fact that there are nine conditional equations used to define the functionality of the method. It is not simply a case of moving the (x,y) coordinates to incorporate the new point, but also the width and height have to be adjusted to encapsulate both the point and the rectangle using the minimum possible area. As can be seen we need many equations with many repeated computations to define the functionality of this method. However as we will see in the next method we will look at, there are ways we can reduce the number of equations required and to make them easier to write.

```
/**
 *
 * @param r Rectangle2D
 * @return Rectangle2D
 * <code>
 * var R2 : Rectangle2D .
 * eq (R).createUnion(R2)q = Rectangle2D(min((R).x,(R2).x),
 *    min((R).y,(R2).y),max((R).x + (R).width,(R2).x +
 *    (R2).width) - min((R).x,(R2).x),max((R).y +
 *    (R).height,(R2).y + (R2).height) - min((R).y,
 *    (R2).y)) .
 * </code>
 */
public Rectangle2D createUnion(Rectangle2D r){
  ...
}
```

The above method is similar to the *add* method. In this case is it returns the union of two *Rectangle2D* instances. If we were to write equations like we did for the *add* method it would take many conditional equations. For this method we predefine a couple of functions that allows us to reduce the code down to one unconditional equation.

```
eq (R).createUnion(R2)q = Rectangle2D(min((R).x,(R2).x),
    min((R).y,(R2).y),max((R).x + (R).width,(R2).x +
    (R2).width) - min((R).x,(R2).x),max((R).y +
    (R).height,(R2).y + (R2).height) - min((R).y,
    (R2).y)) .
```

The two functions we define are *min* and *max* which return the minimum and the maximum of two integers respectively.

```
op min : Int Int -> Int .
op max : Int Int  -> Int .

ceq min(I,J) = I if I <= J .
ceq min(I,J) = J if I > J .

ceq max(I,J) = I if I >= J .
ceq max(I,J) = J if I < J .
```

These functions are not linked to a class specification and work as completely separate functions. They are defined with the built in algebra module of our Maude code. However it could be defined within the *Rectangle2D* class specification's *hidden* section as this part of the class specification is passed into the Maude code unchanged. However we chose not to do this as we feel functions like *min* and *max* are useful general tools which many classes would find useful to have access to. The *add* method needed nine equations due to it having to deal with different cases of two numbers being maximal or minimal. The *createUnion* method is essentially similar. By introducing the *min* and *max* functions we can reduce the equations down to just one. We however left *add* using nine equations to demonstrate the difference.

Finally we will look at *isEmpty*.

```
/**
 *
 * @return boolean
 * <code>
 * ceq (R).isEmpty()q = true if
 *     ((R).width <= 0) or ((R).height <= 0) .
 * eq (R).isEmpty()q = false [owise] .
 * </code>
 */
public boolean isEmpty(){
   ...
}
```

As we are now dealing with an actual two dimensional shape we can check to see if the rectangle is empty or not.

```
ceq (R).isEmpty()q = true if ((R).width <= 0) or
   ((R).height <= 0) .
eq (R).isEmpty()q = false [owise] .
```

A *Rectangle2D* instance is empty if either its width or height is less than or equal to zero.

There are several functions that we are unable to model in this class. There are two *getPathIterator* methods that we are unable to model as we do not model the *AffineTransfrom* classes which they use. We have not modelled *hashCode* and *toString* for the same reasons we gave in other examples. The two *intersectsLine* methods were not modelled as due to the fact our example is limited to just integers we felt that the methods' results would be too inaccurate to make it worth modelling them. Finally we are unable to model the two *outcode* methods. This is due to the fact that they deal with binary operations such as *OR* and as yet our model does not include binary operations.

6.5 An Inheritance Example

We will now look at a series of classes that demonstrate inheritance modelling. We have defined three classes: *Person*, *Student*, and *PostGrad*. The *Person* class defines some basic details for defining a general person such as age and name. The *Student* class inherits from *Person* and expands on it with student specific details such as their grade. Finally *PostGrad* inherits from *Student* and hence also inherits from *Person.* It expands on these with Postgraduate specific details.

For this example we will look at how the *PostGrad* class is modelled algebraically. We will look at aspects from all three classes and then look at the code generated to model them in the *PostGrad* FAS

The subclass structure of the inherit classes is defined using subsorts as follows.

```
subsort Student < Person .
subsort PostGrad < Student .
```

First we will look at the *toString* method in *Person*

```
/**
 *
 * @return String
 * <code>
 * var P : Person .
 * eq (P).toString()q = (P).title + " " + (P).name .
 * </code>
 */
public String toString(){
  return title + " " + name;
}
```

This method returns a string consisting of the person's title and name. When *Student* inherits this method it provides its own definition and thus overrides the *Person*'s *toString* method.

```
/**
 *
 * @return String
 * <code>
 * var S : Student .
```

```
   * eq (S).toString()q = (S).title + " " + (S).name +
   *    " grade:" + (S).grade .
   * </code>
   */
  public String toString(){
    return title + " " + name + " grade:" + grade;
  }
```

This new *toString* method now also returns the student's grade as well as their title and name. The *PostGrad* class does not have its own definition of *toString* so it does not override the method when it inherits it from *Student*. The operators for this method in *Postgrad* are as follows.

```
op _.toString() : PostGrad  -> PostGradString .
op _.toString() : PostGradString  -> PostGradString .
op _.toString()o : PostGrad -> PostGrad .
op _.toString()q : PostGrad -> String .

op _.super.toString() : Student  -> StudentString .
op _.super.toString() : StudentString  -> StudentString .
op _.super.toString()o : Student -> Student .
op _.super.toString()q : Student -> String .

op _.super.toString() : PostGrad  -> PostGradString .
op _.super.toString() : PostGradString  -> PostGradString .
op _.super.toString()o : PostGrad -> PostGrad .
op _.super.toString()q : PostGrad -> String .

op _.super.super.toString() : Person  -> PersonString .
op _.super.super.toString() : PersonString  -> PersonString .
op _.super.super.toString()o : Person -> Person .
op _.super.super.toString()q : Person -> String .
```

The above defines the operators for the various *toString* methods and their *super* versions together with different versions to accept all the possible different class instance tuple input types.

```
eq (P).toString()q = (P).title + " " + (P).name [owise]  .
eq (S).toString()q = (S).title + " " + (S).name +
    " grade:" + (S).grade [owise] .

eq (P).super.toString()q = (P).title + " " + (P).name [owise] .
eq (S).super.toString()q = (S).title + " " + (S).name +
    " grade:" + (S).grade .
eq (P).super.toString()q = (P).title + " " + (P).name [owise] .

eq (P).super.super.toString()q = (P).title + " " + (P).name .
```

The above equations define the behaviour of the various *toString* methods and their *super* methods.

```
eq (SYS3).super.toString()q = (SYS3).toString()q [owise] .
eq (SYS3).super.toString()o = (SYS3).toString()o [owise] .

eq (SYS3).super.super.toString()q =
    (SYS3).super.toString()q [owise] .
eq (SYS3).super.super.toString()o =
    (SYS3).super.toString()o [owise] .
```

The above equations are used to reduce a *super* version *toString* to its next most current definition (I.E. if it is the *super.super.toString* method then it gets rewritten as *super.toString*). The *owise* notation is used to ensure this only happens if the method has not been overwritten (I.E. there is no alternative definition to *super.super.toString* from *super.toString*).

```
eq (SYS7).toString() = (SYS7).toString()o,(SYS7).toString()q .
eq (SYS9).toString() =
    (oval(SYS9)).toString()o,(oval(SYS9)).toString()q .

eq (SYS6).super.toString() = (SYS6).super.toString()o,
    (SYS6).super.toString()q .
eq (SYS11).super.toString() = (oval(SYS11)).super.toString()o,
    (oval(SYS11)).super.toString()q .
eq (SYS7).super.toString() = (SYS7).super.toString()o,
    (SYS7).super.toString()q .
eq (SYS9).super.toString() = (oval(SYS9)).super.toString()o,
    (oval(SYS9)).super.toString()q .

eq (SYS5).super.super.toString() =
    (SYS5).super.super.toString()o,
    (SYS5).super.super.toString()q .
eq (SYS13).super.super.toString() =
    (oval(SYS13)).super.super.toString()o,
    (oval(SYS13)).super.super.toString()q .
```

Finally the above equations are used to extract the query and state change parts of the class instance tuple and recombine the result back into a query and state change tuple.

Finally we will look at the *updateGrade* method

```
  /**
   *
   * @param newgrade String
   * @return String
   * <code>
   * eq (S).updateGrade(G)o = (S).grade:=(G) .
   * eq (S).updateGrade(G)q = (S).grade .
   * </code>
   */
  public String updateGrade(String newgrade){
    String oldgrade=this.grade;
    this.grade=newgrade;
    return oldgrade;
  }
```

This method is first introduced in the *Student* class. It has equations that define both a query return value and a state change value.

```
eq (S).updateGrade(G)o = (S).grade:=(G) .
eq (S).updateGrade(G)q = (S).grade .
```

This method updates the old value of the *grade* field and returns the old *grade* value as its query return value.

There is no overriding of this method in *PostGrad* so it is passed into *PostGrad* unchanged.

```
op _.updateGrade(_) : PostGrad  String -> PostGradString .
op _.updateGrade(_) :
    PostGradString  String -> PostGradString .
op _.updateGrade(_)o : PostGrad String -> PostGrad .
op _.updateGrade(_)q : PostGrad String -> String .

op _.super.updateGrade(_) :
    Student  String -> StudentString .
op _.super.updateGrade(_) :
    StudentString  String -> StudentString .
op _.super.updateGrade(_)o : Student String -> Student .
op _.super.updateGrade(_)q : Student String -> String .
```

The above defines the operators for *updateGrade* and its *super* version. Note that as *updateGrade* is inherited from *Student* but not *Person* there is no *super.super.updateGrade* method.

```
eq (S).updateGrade(G)o = (S).grade:=(G) [owise] .
eq (S).updateGrade(G)q = (S).grade [owise] .

eq (S).super.updateGrade(G)o = (S).grade:=(G) .
eq (S).super.updateGrade(G)q = (S).grade .
```

The above equations define the functionality of the *updateGrade* and its *super* version.

```
eq (SYS3).super.updateGrade(SYS4)q =
    (SYS3).updateGrade(SYS4)q [owise] .
eq (SYS3).super.updateGrade(SYS4)o =
    (SYS3).updateGrade(SYS4)o [owise] .
```

The above equations allow us to convert a *super.updateGrade* to a *updateGrade* call. Due to the *owise* they are only called if there isn't already a definition for *super.updateGrade.*

```
eq (SYS7).updateGrade(SYS14) = (SYS7).updateGrade(SYS14)o,
    (SYS7).updateGrade(SYS14)q .
eq (SYS9).updateGrade(SYS14) = (oval(SYS9)).updateGrade(SYS14)o,
    (oval(SYS9)).updateGrade(SYS14)q .

eq (SYS6).super.updateGrade(SYS15) =
    (SYS6).super.updateGrade(SYS15)o,
    (SYS6).super.updateGrade(SYS15)q .
eq (SYS11).super.updateGrade(SYS15) =
    (oval(SYS11)).super.updateGrade(SYS15)o,
    (oval(SYS11)).super.updateGrade(SYS15)q .
```

Finally the above equations allow us to extract the components of a class instance tuple passed to the *updateGrade* method and its *super version* and put the results back together as a new tuple.

6.6 A Reflection Example

In this section we will look at an example of reflection. We have created a simple class called *Book* with two String fields and two methods. The fields are called *name* and *author* and the methods are called *setName* and *getName.*

First we will look at the *name* field. The declaration for this field in the java code is as follows.

```
public String name;
```

The reflection *Field* instance for this field is defined as follows.

```
op nameBookField : -> Field .

eq nameBookField = ("name","String","Book") .
```

This defines the *Field* instance for *name* as having the string name *name*, field type *String*, and that the field belongs to the *Book* class.

Next we will look at the *setName* method. The Java code for this method is as follows.

```
/**
 *
 * @param name String
 * <code>
 * var B : Book .
 * eq (B).setName(N)q = (B).name .
 * eq (B).setName(N)o = (B).name:=(N) .
 * </code>
 */
public String setName(String name){
  String tempname;
  tempname=this.name;
  this.name=name;
  return tempname;

}
```

This method sets the *name* field to a new value and returns the old value of *name*. The reflection *Method* instance for this field is defined as follows.

```
op setNameBookMethod : -> Method .

eq setNameBookMethod = ("setName",add(EStringArray,"String"),
    "String","Book") .
```

This defines the *Method* instance for *setName* as having the string name *setName*, the only input type of *String*, the return type of *String*, and that the method belongs to the *Book* class.

Finally for *setName* we define the equation for the *Method*'s *invoke* method.

```
eq ("setName",add(EStringArray,"String"),"String","Book")
    .invoke(O,OA) = (O).setName(OA[0]) .
```

The *invoke* method invokes the *setName* method by passing to it an instance of a *Book* as *O* and an input string in the first position of the *ObjectArray OA*.

Finally we define the reflection *Class* instance for the *Book* class.

```
eq BookClass = ("Book",add(add(EMethodArray,setNameBookMethod),
    getNameBookMethod),add(add(EFieldArray,nameBookField),
    authorBookField)) .
```

This defines the *Class* instance as having the string *Book*, an array of *Method* instances for the methods *setName* and *getName*, and an array of *Field* instances for the fields *name* and *author* .

6.7 Problems and Future Work

Finally we will examine certain Java functionality that we cannot as yet model. We will discuss the difficulties they present and how in future work we hope to be able to implement these problems.

6.7.1 Exceptions

A major aspect of the Java language that we are unable to model are *Exceptions.* Consider the following example.

```
public String readLine(SomeInput SI) throws Exception
```

The above example is a Java method that we will assume reads a line of data from the SomeInput class instance and returns a *String.* However this method can also throw an exception (Java can throw different types of exceptions such as an *IOException* which relates to input/output exceptions but for the sake of simplifying the example we will just have the method throw the general *Exception* class). In this case the method will stop executing and throw an *Exception* instance. In Java this would either be thrown again by the method that called *readLine* or it will be caught and dealt with as shown in the following example.

```
try{
    String S;
    for(i=0;i<10;i++}{
        namearray[i]=readLine(in);
    }
    S = namearray[10];
}catch(Exception e){
...some code to deal with the exception event
}
```

It should also be noted that the above example could also throw an exception if there is no array item at index 10.. However array out of bounds exceptions are a special type of exception that do not need to be caught and therefore do not fall within the standard exception behaviour. As our model does not model exceptions, if an exception occurs (such as an array out of bounds exception), then the term causing the exception will fail to evaluate as it will not match any equations that are able to evaluate the term. Although exceptions are an important part of the Java language, we do not feel that this compromises the validity of our model. There are many important features in Java and it would be impossible to model them all within the scope of this thesis. Also we are not alone in choosing not to model exceptions as can be seen in the work of Henkel [Hen04].

Exceptions present a complex modelling problem. You cannot naively add an equation for the *readLine* method that returns an *Exception* instance in the event of a problem as this will not match the type that is expected to be returned from *readLine.* The modelling of exceptions algebraically is considered to be a difficult and a lot of work has been devoted to this topic by other researchers [GTW78, BT88]. In future work we suggest redesigning the internal equations of the class to be able to cope with exceptions. One suggestion on how to do this is as follows.

In the above example, *readLine* could be adapted so that instead of returning a *String* as defined above it would return a tuple which could consist of the return type, an exception, and a boolean which would state whether or not an exception had occured. Extra equations would need to be generated to handle this when it is returned to the calling method and extract the appropriate part of the tuple (either the result part or the exception part depending on the value of the boolean) and then the appropriate equations and operators could be used to process that the result. This would need to be done for all methods and would be part of the internal working of the system, hidden from the user. With care all these extra equations and tuples could be automatically generated so as far as such a user would be concerned, the specification would be behave as they would expect the Java code to behave. The user would be unaware of the tuples and their automatic processing going on internally. This would require a large amount of work to implement within our model and as such is outside the scope of this thesis. At present this solution is only a proposal and would require further work to ascertain its viability.

6.7.2 Static Methods

At present we do not model *static* methods. Static methods are effectively "real" functions in that they do not rely on the class they belong to being instantiated in order to run the method. This would be simple to implement and is only missing from our model due to time restrictions. We have provided in our algebraic class specifications a section called *Operations* which we envision as storing *static* method operations. The only difference a *static* method would require to a normal method is that as well as the standard ability to call it on an class instance we would also need an operator that would allow us to call it on an class name (e.g. `ClassName.methodName(inputs)`). Static methods would only have a query return type as they are unable to change the state of class instances.

6.7.3 Additional Reflection Functionality

Although we are able to model reflection and have elements of the reflection classes already defined there is still much that has not been modelled in this area. Again this is due more to time restrictions than any difficulty in modelling the concepts and it is hoped that in the future more functionality could be implemented in this area of the model. A prime example would be implementing the method in the *Field* class that allows a user to set the value of the corresponding field in a similar manner to how we implemented *invoke* in the *Method* class.

6.7.4 Type Resolution

Our model at present also has problems with resolving types when the classes get sufficiently complex with respect to their inherited methods and fields. However we feel that this problem is caused by the Maude method of resolving types rather than in our model itself. In general the subtype system works well but there are occasional problems that occur when the Maude rewrite engine selects an inherited method for the super type class instance rather than the correct method for the actual type of the class method. For example suppose we have a two classes AClass and BClass, and BClass inherits from AClass. Suppose we do the following in Java.

```
AClass a= new BClass();
```

In the above example we have used a variable of type AClass and created a class instance using BClass' constructor. In Java it will therefore treat the variable a as being of type BClass as it was instantiated using *BClass*'s constructor. Our model does correctly model this.

However let us now assume that AClass had a method called meth and BClass overrides that method with its own definition. We do the following in Java:

```
a.meth();
```

In Java because we initialised a using BClass' constructor then BClass' overridden definition of meth should be called. However in an algebraic specification in Maude AClass' original definition of meth is called which is incorrect behaviour.

This problem would be solvable by imposing a precedence on types and making sure Maude resolves types as we would expect Java to. This would require us to impose our own type resolution rather than use Maude's default type resolution. It is hoped that all of the extra information that Maude would need to do this can be automatically generated. Again it should be noted that the problems to do with type resolution is a problem relating specifically to the way Maude resolves types and is not necessarily a problem with our algebraic model.

6.7.5 Execution Model

Currently, there is no model of execution as discussed in Section 2.1. At present our model only allows us to use the equations to do rewrite tests to spot check the local functionality of individual classes and their instances (occasionally with some "helper" classes and instances added in if needed to fully test the behaviour of the class being modelled). Our model only models the local behaviour of classes and does not model the concept of a program. In order to model a program the implementation of an execution model would be needed. This would be quite complex and is beyond the scope of this thesis.

In order to create an execution model we would need to define our own rewrite strategy. At present we use the Maude default rewrite strategy. In order to define our own we would need to make use of Maude's meta level functionality. In object-oriented programming the state of a program is the set of all possible states of all the class instances of an object-oriented program. The state of the class instances can be considered to be the current states of each of their individual fields. This would require us to introduce the concept of repositories for classes and class instances. This would not be too hard to implement.

However, defining a single time step in the system is more difficult. The execution of a method in one class instance may require the execution of many other methods in other class instances before it can complete. With an imperative program the concept of modelling the execution is relatively easy where all the variables in the program can be used to define the current state of the system and a function can be used to evaluate the next step in the program . With object-oriented programs this is much harder as you have a group of class instances, each with their own set of variables (fields). Therefore further work would need to be done to decide what constitutes a step in our model of object-oriented programs. As stated above, in order to solve this problem we would need to define our own rewrite strategy. A new strategy would need to decide which equations to use to evaluate terms for a given time step and would also need to update the class instance repository with new class instances, updated class instances, and remove old class instances that are no longer needed. The default Maude rewrite strategy is inadequate for this task.

All of this presents a complex problem in modelling execution that is beyond the scope of this thesis to solve. We feel that what we have contributed is a set of operations and equations that would link in with an execution model which would use a special rewrite strategy to select which equations it needed to use to calculate the next step and call each equation in the correct order. The execution model would need to have some concept of what is a valid complete step and what would be viewed as a substep in a computation of an overall valid step.

6.7.6 Input and Output

Another problem with the specification is that there is at present no way of handling input and output and thus Java's input and output classes cannot be specified. Again this comes down to a question of time and with future work this could be successfully implemented. This could be achieved by modelling input and output as streams, where the stream is defined as providing certain data at specific steps in a system. A stream would be defined in general as follows.

$$A : T \to B$$

Where a stream A at a given step $t \in T$ provides a value $b \in B$. For example $A[10]$ would return the value of the stream at step 10 (for instance T could be the time in seconds, so intuitively $A[10]$ would return the value of the stream at 10 seconds from the initial starting point). In order to do this we would need to have an execution model as discussed in section 6.7.5. However input particularly is a difficult concept to model.

6.8 Sources

The majority of the examples in this section are taken from the Java 1.4 API [Sun05g]. In places we have, as discussed earlier, made adaptations to the classes in order that we can more effectively demonstrate our modelling techniques. Any classes that are not part of the Java API were written by ourselves usually to demonstrate a specific feature of the modelling process. The FASs of the classes have mostly been at least partly generated by our automated conversion tool as discussed in Section 5.3. This was done in this way because to manually create the FASs takes a lot of book keeping and work as discussed earlier.

Chapter 7

Conclusions

In this thesis we have algebraically specified some of the functionality of Java classes using order-sorted specifications. We have shown a methodology that will allow other programmers to easily be able to define specifications for their own Java classes. We have kept the modelling of Java as broad as possible to try and cover a wide ranging specification of Java's functionality. We have shown how to model specialised functionality of Java such as the important Reflection API. We have also shown how the generation of many of the equations defining the functionality of Java classes can be automated thus reducing the work in specifying new classes.

In Chapter 2 we looked at the background to our research, examining object-oriented programming and algebraic specification. Most importantly we looked at how we can equate order-sorted algebra to many-sorted algebra which allowed us to use subtypes to model inheritance as a notational convenience.

In Chapter 3 we examined the basic structure of a Java class and identified what we considered to be the key features of the class. We then defined the structure of an Algebraic Class Interface and added semantic equations to the interface to create an Algebraic Class Specification (ACS). The ACS is not a complete specification and is designed to be provide us with a more human readable model, consisting of only those parts that cannot be programmatically inferred from the Java language definition. We showed with the aid of examples, the complete process in generating an ACS from a Java class. We then looked at how our work related to and built upon the work of [STR03]. Finally we gave an overview of the code we have written that can build ACSs and FASs from Java classes with extra embedded information. In this chapter we focussed primarily on the stages of harvesting information from the Java class and generating the ACS. Due to the length of the code, we discussed in general the implemented algorithm and showed examples of the code for the particular case of generating ACS specifications for the fields of a Java class.

We then looked in Chapter 4 at how create a Full Algebraic Specification (FAS) written in Maude. Writing the specification in Maude allows it to be executable which we consider to be an important advantage (though note that there is not universal agreement about this). The FAS contains all the extra operations and equations needed to model those parts of a class that are not included in the ACS (i.e. those parts that are part of the implicit Java language definition). We examined how we create an FAS for classes with inheritance and examined all the extra equations and operators that needed to be generated to support this. We again examined how our work in this chapter related to and built upon the work of [STR03]. We looked particulary at interface tagging, joining and flattening which was closely related with how we dealt with inheritance in FASs. Finally we again discussed the code we had written for building ACSs and FASs from Java classes. In this chapter we primarily focussed on the building of the FAS using the information gathered from the Java class as discussed earlier in Chapter 3. We again gave a general overview of the implemented algorithm and then showed examples of the code for the particular case of generating FAS specifications for the fields of a Java class.

In Chapter 5 we looked at the functionality we had pre-defined in our model, specifically functionality which could not be easily defined using the specification process shown in Chapters 3 and 4. The aim of this chapter was to provide the user with some of the key functionality that comes built into the Java API. We looked at how we modelled arrays and how to generate equations for arrays. We then looked at how we modelled the Java Reflection API and how we solved specifying core components of these complex and unique classes. Finally we looked at the Algebraic Specification Generator (ASG) that we had written which allowed us to add and test new functionality to our model. The ASG required that only the semantics of methods and constructors of classes needed to be provided by the user by means of equations and conditional equations. The ASG would be able to generate all the other structure equations and operators that defined an FAS of a class as discussed in Chapter 4.

Finally in Chapter 6 we examined a wide range of example class specifications looking in particular at the process of taking a Java class and producing an FAS. The aim here was to demonstrate our specification in action over a series of varied examples. We looked at both built in Java API classes and also user defined classes, examining interesting modelling problems. Finally we examined functionality that at present we cannot model and offered suggestions as to how future work could address these short comings.

7.1 Future Work

We have already discussed future work on modelling Java functionality in section 6.7 so we will not do so again here. In this section we will look at future work in building supporting software that does not specifically relate to Java functionality. The majority of suggestions in this section focus on how to make our specification techniques easier to use for programmers by making the model easier to read and also by trying to reduce the amount of information a user has to provide to build a model of their system.

One area where the specification process could be improved is in the query/command structure of methods. At present a user has to state whether they are defining the query or command part of the semantics of a method. It would be desirable for the user not to have specify which part they are defining of a method and for the specification process to be able to infer which part they are defining and generate the appropriate equations. The first step towards this would be to allow a default where if the user does not specify whether they are defining the query or command part of a methods' semantics, the tool will automatically assume it to be one of them. To actually be able to automatically differentiate between the two types a more intelligent modelling process needs to be developed as at present the technique is largely syntactic.

The actual process of generating equations for the FAS also needs reviewing as it is likely that several redundant equations and operators are generated for each FAS. Although this does not affect our mathematical model of class specifications, a more efficient and easier to understand set of equations would be desirable.

As mentioned elsewhere in the thesis, although we embed our formal semantic equations for the methods and constructors within Javadoc comments so as they can appear in API documentation, this is at present without any proper formatting and is incorrectly displayed within the actual documentation. Future work needs to change the tags we use to the correct Javadoc tags that will allow the information to be properly presented.

Finally, at present our model specifically works with Java. With several object-oriented programming languages in common use such as C++ and C# it would be desirable to adapt our model to work with these other languages. As shown in this thesis, our model is specifically aimed at Java. However it is sufficiently general in places when it refers to object-oriented concepts and features that it could be adapted to other object-oriented languages. We therefore believe this could be done without a radical rewrite of the model and the modelling process.

Bibliography

[AA86] G. Anderson and P. Anderson. *The UNIX C Shell Field Guide.* Prentice-Hall, 1986.

[ABB+03] L. Andrade, P. Baldan, H. Baumeister, R. Bruni, A. Corradini, R. De Nicola, J. L. Fiadeiro, F. Gadducci, S. Gnesi, P. Hoffman, N. Koch, P. Kosiuczenko, A. Lapadula, D. Latella, A. Lopes, M. Loreti, M. Massink, F. Mazzanti, U. Montanari, C. Oliveira, R. Pugliese, A. Tarlecki, M. Wermelinger, M. Wirsing, and A. Zawlocki. Agile: Software architecture for mobility. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends In Algebraic Techniques*, volume 2755 of *Lecture Notes In Computer Science*, pages 1–33, 2003.

[ABBK99] E. Astesiano, B.Krieg-Bruckner, and H.-J. Kreowski. *Algebraic Foundations Of System Specification.* IFIP State-of-the-Art Reports. Springer Verlag, 1999.

[ABR99] E. Astesiano, M. Broy, and G. Reggio. Algebraic specification of concurrent systems. In E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors, *IFIP WG 1.3 Book On Algebraic Foundations Of System Specification.* Springer Verlag, 1999.

[ACI05] Apple Computer Inc. http://www.apple.com/macosx/, 20th September 2005.

[AGH00] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language (3rd Edition).* Addison-Wesley Professional, 3rd edition, 2000.

[AH00] S. Antoy and R. G. Hamlet. Automatically checking an implementation against its formal specification. *Software Engineering*, 26(1):55–69, 2000.

[AL97] M. Abadi and R. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory And Practice Of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214, pages 682–696. Springer-Verlag, 1997.

[AMRW85] E. Astesiano, G.F. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent systems. In H. Erhig, C. Floyd, M. Nivat, and J. Thatcher, editors, *TAPSOFT'85, Vol. 1*, volume 185 of *Lecture Notes in Computer Science*. Springer, 1985.

[And64] C. Anderson. *An Introduction To Algol 60.* Addison-Wesley, 1964.

[AR02] M. Adelaide and O. Roux. A class of decidable parametric hybrid systems. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology*, volume 2242 of *Lecture Notes In Computer Science*, pages 132–146, 2002.

[ARZ99] E. Astesiano, G. Reggio, and E. Zucca. Stores as homomorphisms and their transformations - a uniform approach to structured types in imperative languages. In *Science Of Computer Programming*, volume 34, pages 163–190. 1999.

[BAN96] M. Burrows, M. Abadi, and R. Needham. A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In *William Stallings, Practical Cryptography For Data Internetworks, IEEE Computer Society Press, 1996.* 1996.

[Bau05] H. Baumeister. http://www.pst.informatik.uni-muenchen.de/projekte/agile/, 20th September 2005.

[BBM03] P. Baldan, R. Bruni, and U. Montanari. Pre-nets, read arcs and unfolding: A functorial presentation. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends In Algebraic Techniques*, volume 2755 of *Lecture Notes In Computer Science*, pages 145–164, 2003.

[BCC$^+$wn] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan M. Leino, and E. Poll. An overview of JML tools and applications. In *International Journal On Software Tools For Technology Transfer.* Springer, Unknown.

[BCDS02] G. Barthe, P. Courtieu, G. Dufay, and S. M. De Sousa. Tool-assisted specification and verification of the JavaCard platform. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology*, volume 2242 of *Lecture Notes In Computer Science*, pages 41–59, 2002.

[Ber86] J. T. Berry. *Advanced C programming.* Prentice Hall Press, 1986.

[Bir72] G. M. Birtwistle. *Simula Begin.* Petrocelli/Charter, 1972.

[BJM00] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1–2):35–132, 2000.

[BM04] M. Bidoit and P. D. Mosses. CASL *User Manual.* LNCS 2900 (IFIP Series). Springer, 2004.

[BMMS01] R. Bruni, J. Meseguer, U. Montanari, and V. Sassone. Functorial models for petri nets. *INFCTRL: Information And Computation (formerly Information And Control)*, 170, 2001.

[BT87] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data types. In *Theoretical Computer Science*, number 50, pages 137–181, 1987.

[BT88] J. A. Bergstra and J. V. Tucker. The inescapable stack: An excercise in algebraic specification with total functions. Report 13.88, The University of Leeds, Centre For Theoretical Computer Science, 1988.

[BT93] J. A. Bergstra and J. V. Tucker. Equational specifications for computable data types: 6 hidden functions suffice and other sufficiency bounds. In J. V. Tucker and K. Meinke, editors, *Many Sorted Logic And Its Applications*, pages 89–102. J Wiley and Sons, 1993.

[CAF97] The Common Algebraic Framework Initiative. CASL The CoFI algebraic specification language rationale, May 1997. Obtainable from http://www.brics.dk/Projects/CoFI/.

[CAF05a] The Common Algebraic Framework Iniative. http://www.brics.dk/Projects/CoFI/Documents/CASL/Sample/, 20th September 2005.

[CAF05b] The Common Algebraic Framework Initiative. http://www.brics.dk/Projects/CoFI/, 20th September 2005.

[CAF05c] The Common Algebraic Framework Initiative. http://www.brics.dk/Projects/CoFI/CASL.html, 20th September 2005.

[CC99] S. Cimato and P. Ciancarini. A formal approach to the specification of Java components. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques For Java Programs*, pages 18–25, 1999.

[CDE+01] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[CDE+03] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques And Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[CDE+04] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual (Version 2.1)*. SRI International and Department of Computer Science Univeristy of Illinois at Urbana-Champaign, March 2004. Obtainable from http://maude.cs.uiuc.edu/maude2-manual/.

[CDH+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, Corina S. Păsăreanu, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *International Conference On Software Engineering*, pages 439–448, 2000.

[CELM00] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Electronic Notes In Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[Cla99] T. Clark. Formal refinement and proof of a small Java program. In B. Jacobs, G. T. Leavens, P. Müuller, and A. Poetzsch-Heffter, editors, *Formal Techniques For Java Programs*, pages 26–32, 1999.

[CN91] B. Cox and A. Novobilski. *Object-Oriented Programming : An Evolutionary Approach.* Addison Wesley, 2nd edition, 1991.

[DF94] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. In *ACM Transactions On Software Engineering And Methodology*, pages 101–130. ACM Press, 1994.

[Dha97] K. K. Dhara. Behavioral subtyping in object-oriented languages. Technical Report 97-09, Iowa State University, Department of Computer Science, 1997.

[DL96] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings Of The 18th International Conference On Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, 1996.

[DM99] F. Durán and J. Meseguer. The Maude specification of Full Maude, 1999. Manuscript, SRI International. Available at http://maude.csl. sri.com, February 1999. 28.

[DMN70] O.-J. Dahl, B. M., and K. Nygaard. *Common Base language*, volume S-22 of *Norwegian Computing Centre. Publication.* Norwegian Computing Centre, revised edition, 1970.

[Duf97] J.-F. Duford. Algebras and formal specifications in geometric modelling. In *The Visual Computer*, chapter 13, pages 131–154. Springer-Verlag, 1997.

[EM85] H. Erhig and B. Mahr. Fundamentals of algebraic specification I: Equations and initial semantics. In *EATCS Monograph*, volume 6. Springer-Verlag, 1985.

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[ESo05] Eiffel Software. http://archive.eiffel.com/doc/manuals/technology/contract/, 20th September 2005.

[FF04] I. R. Forman and N. Forman. *Java Reflection In Action.* Manning, 2004.

[Fia02] J. L. Fiadeiro. Application support for service-oriented architecture. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology*, volume 2242 of *Lecture Notes In Computer Science*, pages 75–82, 2002.

[Fis99] C. Fischer. Software development with Object-Z, CSP and Java: A pragmatic link from formal specifications to programs. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques For Java Programs*, pages 33–39, 1999.

[GD94] J. A. Goguen and R. Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures In Computer Science*, 4(3):363–392, 1994.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. In *Acta Informatica*, volume 10, pages 27–52, 1978.

[GHG$^+$93] J. V. Guttag, J. J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J. M. Wing. *Larch: Languages And Tools For Formal Specification.* Texts and Monographs in Computer Science. Springer, 1993.

[GLR03] J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends In Algebraic Techniques*, volume 2755 of *Lecture Notes In Computer Science*, pages 216–232, 2003.

[GM92] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[GM96] J. Goguen and G. Malcolm. *Algebraic Semantics Of Imperative Programs.* MIT Press, 1996.

[GM00] J. Goguen and G. Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, 2000.

[Gog78] J. Goguen. Order sorted algebra. Semantics and Theory of Computation Series 14, UCLA Computer Science Department, 1978.

[Gos96] J. Gosling. *The Java Application Programming Interface: Window Toolkit and Applets*, volume 2. Addison Wesley, 1996.

[Gre05] D. Green. Trail: The Reflection API, 20th September 2005. Obtainable from
http://java.sun.com/docs/books/tutorial/reflect/.

[GRV03] J. Gosling, M. Robinson, and P. Vorobiev. *Swing Second Edition.* Manning, 2nd edition, 2003.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends In Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[GTWW77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *JACM*, page 69, January 1977.

[GW88] J. Goguen and T. Winkler. Introducing OBJ3. Technical report, SRI International, Computer Science Lab, 1988.

[GWM+93] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications Of Algebraic Specification Using OBJ.* Cambridge, 1993.

[Har89] N. A. Harman. *Formal Specifications For Digital Systems.* PhD thesis, Univeristy of Leeds, School of Computer Studies, 1989.

[Har00] N. A. Harman. Correctness and verification of hardware systems using Maude. Technical Report 3, University of Wales, Department of Computer Science, 2000.

[Har02] N. A. Harman. Verifying a simple pipelined microporcessor using Maude. In M. Cerioli and G. Reggio, editors, *Recent Trends In Algebraic Techniques*, volume 2267 of *Lecture Notes In Computer Science*, pages 128–151, 2002.

[HD03] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference.* Springer, 2003.

[HD04a] J. Henkel and A. Diwan. Case study: Debugging a discovered specification for java.util.ArrayList by using algebraic interpretation. Technical Report CU-CS-970-04, University of Colorado at Boulder, 2004.

[HD04b] J. Henkel and A. Diwan. A tool for writing and debugging algebriac specifications. In *International Conference On Software Engineering (ICSE)*, 2004.

[Hen96] T. Henzinger. The theory of hybrid automata. In *IEEE Symposium On Logic In Computer Science*, pages 272–282. 1996.

[Hen04] J. Henkel. *Discovering And Debugging Algebraic Specifications For Java Classes.* PhD thesis, University of Colarado, 2004.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, 1969.

[Hof02] P. Hoffman. Verifying architectural specifications. In M. Cerioli and G. Reggio, editors, *Recent Trends In Algebraic Techniques*, volume 2267 of *Lecture Notes In Computer Science*, pages 152–175, 2002.

[Hof03] P. Hoffman. Verifying generative CASL architectural specifications. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends In Algebraic Techniques*, volume 2755 of *Lecture Notes In Computer Science*, pages 233–252, 2003.

[HS96] M. Hughes and D. Stotts. Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In *International Symposium On Software Testing And Analysis.* ACM Press, 1996.

[HW03] J. Hughes and M. Warnier. The coinductive approach to verifying cryptographic protocols. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends In Algebraic Techniques*, volume 2755 of *Lecture Notes In Computer Science*, pages 268–283, 2003.

[HWG03] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language.* Addison Wesley, 2003.

[Jac96a] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in LNCS, pages 210–231. Springer, Berlin, 1996.

[Jac96b] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.

[JML05] JML Modelling Group. http://www.cs.iastate.edu/~leavens/JML/, 20th September 2005.

[JP03] B. Jacobs and E. Poll. Coalgebras and monads in the semantics of Java. In *Theoretical Computer Science*, volume 291, pages 329–349. Elsevier, 2003.

[JSY03] B. Johnson, C. Skibo, and M. Young. *Inside Visual Studio .NET.* Microsoft Press International, 2003.

[JvdBH+98] B. Jacobs, J. van den Berg, Ma. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *Proceedings, Object-Oriented Programming Systems, Languages And Applications (OOPSLA'98)*, pages 329–340, Vancouver, Canada, 1998.

[Ker88] B. W. Kernighan. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[KK03] H.-J. Kreowski and S. Kuske. Approach-independant structuring concepts for rule-based systems. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends In Algebraic Techniques*, volume 2755 of *Lecture Notes In Computer Science*, pages 299–311, 2003.

[Koc03] S. Kochan. *Programming In Objective C.* Sams, 2003.

[Kra98] R. Kramer. iContract - the Java design by contract tool. In *Technology Of Object-Oriented Languages And Systems.* IEEE Computer Society, 1998.

[KT99] G. Kniesel and D. Theisen. Jac – java with transitive readonly access control, 1999.

[LAGB02] F. Ledoux, A. Arnould, P. Le Gall, and Y. Bertrand. Geometric modelling with CASL. In M. Cerioli and G. Reggio, editors, *Recent Trends In Algebraic Techniques*, volume 2267 of *Lecture Notes In Computer Science*, pages 176–200, 2002.

[Lam02] Y. Lamo. *Institution Of Multialgebras As A General Framework For Algebraic Specification.* PhD thesis, University of Bergen, Department of Informatics, 2002.

[LBR05] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. Technical Report 98-06-rev27, Department of Computer Science, Iowa State University, 2005.

[LC05] G. T. Leavens and Y. Cheon. Design by contract with jml, 2005.

[LPC$^+$05] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual, 2005.

[LW90] G.T. Leavens and W. E. Weihl. Reasoning about oject-oriented programs that use subtypes. Technical Report 90-03B, Iowa State University, Department of Computer Science, 1990.

[LW03] Y. Lamo and M. Walicki. Combining specification formalisms in the 'general logic' of multialgebras. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends In Algebraic Techniques*, volume 2755 of *Lecture Notes In Computer Science*, pages 328–342, 2003.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 2nd edition, 1999.

[McC98] G. McCluskey. Using Java Reflection, January 1998. Obtainable from
http://java.sun.com/developer/technicalArticles/ALT/
Reflection/.

[McC03] T. McCombs. *Maude 2.0 Primer (Version 1.0).* SRI International and Department of Computer Science Univeristy of Illinois at Urbana-Champaign, August 2003. Obtainable from http://maude.cs.uiuc.edu/primer/.

[Mes98] J. Meseguer. Membership algebra as a logical framework for equational specification. In *In 12th International Workshop On Recent Trends In Algebraic Development Techniques (WADT'97)*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mey92a] B. Meyer. Applying 'design by contract'. *Computer (IEEE)*, 25(10):40–51, October 1992.

[Mey92b] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[MG85] J. Meseguer and J. A. Goguen. Initiality, induction, and computation. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[MIT05] MIT. http://www.sds.lcs.mit.edu/spd/larch/, 20th September 2005.

[MPH97] P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik 97: Informatik als Innovationsmotor.* Springer-Verlag, 1997.

[MT92] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook Of Logic In Computer Science*, pages 189–411. Oxford University Press, 1992.

[Mül02] P. Müller. *Modular Specification And Verification Of Object-Oriented Programs.* Number 2262 in Lecture Note in Computer Science. Springer, 2002.

[NCC05] Netscape Communications Corporation. http://browser.netscape.com/nsb/download/default.jsp, 20th September 2005.

[Pol97] R. Pollie. *Write Once, Run Anywhere–Is It For Real?* Obtainable from http://java.sun.com/features/1997/aug/wora.html, August 1997.

[PW88] L. J. Pinson and R. S. Wiener. *An Introduction To Object-Oriented Programming And Smalltalk.* Addison-Wesley, 1988.

[RCA00] G. Reggio, M. Cerioli, and E. Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo Editors, editors, *AMiLP 2000*, number 16, 2000.

[Ree01] D. Ll. L. Rees. *A Theory Of Software Interfaces.* PhD thesis, University of Wales, Department of Computer Science, 2001.

[RL96] K. Rustan and M. Leino. Ecstatic: An object-oriented programming language with axiomatic semantics. Technical report, Digital Equipment Corporation System Reasearch Center, 1996.

[RLS97a] K. Rustan, M. Leino, and R. Stata. Checking object invariants. Technical Report #1997-007, Palo Alto, USA, 1997.

[RLS97b] K. Rustan, M. Leino, and R. Stata. Virginity: A contribution to the specification of object-oriented software. Technical Report #1997-001, Palo Alto, USA, 1997.

[RS00] O. F. Rana and M. S. Shields. Performance analysis of Java using petri nets. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *High Performance Computing And Networking*, number 1823 in Lecture Notes In Computer Science, pages 657–667, 2000.

[Sko02] M. Skoglund. Sharing objects by read-only references. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology*, volume 2242 of *Lecture Notes In Computer Science*, pages 457–472, 2002.

[SM02] L. Schroder and T. Mossakowski. Hascasl: Towards intergrated specification and development of functional programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology*, volume 2242 of *Lecture Notes In Computer Science*, pages 99–116, 2002.

[SM06] R. Sasse and J. Meseguer. Java+ITP: A verification tool based on hoare logic and algebraic semantics. In *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006), Vienna, Austria*, ENTCS. Elsevier, 2006. to appear, see: http://banyan.cs.uiuc.edu/pub/JavaITP.pdf.

[SRI05] SRI International. http://maude.cs.uiuc.edu/, 20th September 2005.

[ST99] D. Sannella and A. Tarlecki. Algebraic preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Bruckner, editors, *Algebraic Foundations Of Systems Specification*, IFIP state-of-the-art reports, chapter 2, pages 13–30. Springer, 1999.

[Ste96] K. Stephenson. *An Algebraic Approach To Syntax, Semantics And Compilation.* PhD thesis, University of Wales Swansea, Department of Computer Science, 1996.

[Ste02] J. G. Stell. A framework for order-sorted algebra. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology*, volume 2242 of *Lecture Notes In Computer Science*, pages 396–410, 2002.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[STR03] K. Stephenson, J. V. Tucker, and D. Rees. The algebraic structure of interfaces. *Science Of Computer Programming*, 49:47–48, 2003.

[Sun05a] Sun Microsystems. http://java.sun.com/, 20th September 2005.

[Sun05b] Sun Microsystems. http://java.sun.com/developer/technicalArticles /Programming/GettingStarted/index.html, 20th September 2005.

[Sun05c] Sun Microsystems. Java Reflection API, 20th September 2005. Obtainable from http://java.sun.com/j2se/1.4.2/docs/api/index.html.

[Sun05d] Sun Microsystems. http://java.sun.com/docs/books/jls/second_edition/html/ syntax.doc.html, 20th September 2005.

[Sun05e] Sun Microsystems. http://java.sun.com/docs/books/tutorial/java/javaOO /classvars.html, 20th September 2005.

[Sun05f] Sun Microsystems. http://java.sun.com/docs/books/tutorial/java/data /arrays.html, 20th September 2005.

[Sun05g] Sun Microsystems. http://java.sun.com/j2se/1.4.2/index.jsp, 20th September 2005.

[Sun05h] Sun Microsystems. http://java.sun.com/j2se/1.5.0/index.jsp, 20th September 2005.

[Ten02] R. D. Tennent. *Specifying Software.* Cambridge University Press, 2002.

[The04] The Common Algebraic Framework Initiative. CASL *Reference Manual.* LNCS 2960 (IFIP Series). Springer, 2004.

[TM94] J. Turner and L. McCluskey. *The Construction Of Formal Specifications : An Introduction To The Model-Based And Algebraic Approaches.* McGraw-Hill international series in software engineering. McGraw-Hill, 1994.

[TS06] J. V. Tucker and K. Stephenson. Data, Syntax and Semantics. In preparation, 2006.

[Tuc06] J. V. Tucker. *Personal Communication.* In preparation, 2006.

[TW95] P. Thomas and R. Weedon. *Object-Oriented Programming In Eiffel.* International computer science series. Addison-Wesley, 1995.

[Wag81] E. G. Wagner. Lecture notes on the algebraic specification of data types. Research Report RC 9203 39787, Mathematical Sciences Center, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1981.

[Wec91] W. Wechler. Universal algebra for computer scientists. In *EATCS Monograph.* Springer-Verlag, 1991.

[Wil02] M. Williams. *Microsoft Visual C# .NET.* Microsoft Press, 2002.

Appendix A

CD Repository

A.1 File Locations

The following files can be found on the CD attached to the inside back cover of this thesis in the following locations.

- The predefined BUILDLINK algebra for FAS Maude module = `inbuilt\`. This consists chiefly of declarations of sorts such as NAT numbers, basic definitions for Integer and String classes, and definitions for specialised classes such as the Reflection classes.
- Example Java classes source files = `examples\classes\`
- Example FAS Maude modules = `examples\fas\`
- Test data for example FASs = `examples\test\`
- Java Translation program source code = `translation\`

A.2 Compiling and Running the Translation Program

1. Make sure you have a copy of the Java 1.4 or higher SDK available from `http://java.sun.com`.
2. Copy the `specification\` and `testrun\` directories in the Java translation program on the CD to a directory of your choice.
3. Switch to your chosen directory.

4. Compile the java program by issuing the command `javac testrun\*.java`.

5. Run the program by issuing the following command `java testrun.Application1 AClass c:\location\AClass.java` where:

   - `AClass` is the name of the class you wish to generate an FAS for.
   - `c:\location\AClass.java` is the filename and full path to the location of `AClass` source codes file.

   Once run the ACS will be in the file `output.txt` in the current directory and the FAS in the file `spec.maude`

A.3 Loading the Executable FAS Maude modules

1. Download Maude which is obtainable from `http://maude.cs.uiuc.edu/download/download.php?category=binaries;target=maude-windows.zip`.

2. Copy the file `algebra.maude` from the `inbuilt\` folder on the CD along with any FAS Maude modules and runtime example data Maude modules that you wish to use into the directory into which the Maude program installed.

3. Start maude by running the `maude.bat` program.

4. Load the `algebra.maude` file by typing `in algebra`.

5. Load the FAS module you wish to test by typing `in spec` where `spec` is the name of the FAS file you wish to load.

6. Load (if required) and test data modules you wish to use by typing `in test` where `test` is the name of the test file you wish to load.

7. You are now ready to begin running reduction tests on your loaded FAS.