A Programmers Guide to MMT

Tony Clark, King's College London

Andy Evans, University of York

Stuart Kent, University of Kent

CHAPTER 1

Introduction

Software systems no longer consist of a small collection of modules whose interactions are easy to control. The use of networking and the increasing speed of hardware means that large distributed systems can implement a diverse range of sophisticated applications.

Methods and tools for software development lag behind advances in hardware and do not address the needs of heterogeneous system development. There have been some advances in recent years, in particular object-oriented methods including the Unified Modeling Language. However, there is increasing evidence that a silver bullet for software development does not exist: no single development technology is sufficient to support the life-cycle of large scale applications.

Each aspect of an application must be modelled and combined to form a complete design. Aspects include the business logic, the component distibution, the information structure and the use cases. Aspects overlap in terms of their information content and must be shown to be consistent.

Development proceeds by transforming different views of a system through a number of increasingly detailed stages. Ultimately a system must be expressed as a collection of interacting implementation modules. Aspect models are translated and merged.

In order for development to succeed as a quality controlled process all features must be modelled including high level aspect models, low level implementation models, consistency checks between models at the same level of abstraction and translations between models at different levels of abstraction.

Our approach advocates treating software development as a series of transformations applied to models. Complexity in development is controlled by expressing large models as a combination of smaller models. Diversity in applications is supported by the development of specific modelling languages at all levels of abstraction during development. Diverse languages are used in consort by modelling consistency relationships that must be preserved by successive development refinements.

To support this approach we have developed a method and technologies that support the method. The technologies have been implemented in terms of a meta-programming environment called MMT. The aim of this book is to provide a primer on MMT from the ground up.

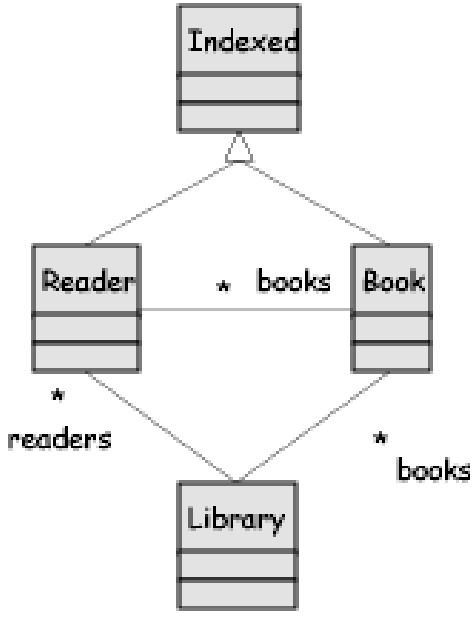
MMT is a reflexive open meta-programming environment in the style of Smalltalk, CLOS and ObjVLisp. It is organised as a virtual machine that runs a small object language. A programming language (called MML) is performed on the VM by translation to the smaller object language. Most features of MMT are written in MML. Since MML is object-oriented its behaviour can be re-defined and extended by the user.

MML provides a collection of classes that support most aspects of main-stream object-oriented development. In addition MML implements features that directly support the MMM method of object design.

This book is organised as a series of chapters. The order of the chapters reflects the layering of MML libraries. The earlier chapters on data values and classes describe essential features of the environment. Later chapters will be of interest to readers who want to use advanced features of MMT, for example to develop new meta-languages or to tailor the graphical toolset.

CHAPTER 2

A Simple Example



At its simplest level MMT can be viewed as an interactive object-oriented programming environment. MML is a complete object-oriented programming language with features including packages, classes, attributes, methods, inheritance, objects and slots. This section provides an introduction to these features by developing a simple application in MML.

The example application is a library that consists of books and readers. New books may be shelved in the library. New readers may be registered with the library. A shelved book may be borrowed by a registered reader in which case the book is removed from the library shelves. When the book is returned, it is replaced on the library shelves.

The application is defined in a file called *Library.pkg*. The contents of this file is developed below together with a commentary; the numbers on the left refer to the line number in the file. MML keywords are in bold font.

```
1. package Lib
```

A *.pkg* file contains a definition, usually a *package*. Line 1 shows the start of the definition of a package called Lib. A package definition consists of a collection of sub-definitions.

```
2.      class Library
```

Line 2 shows the start of a class definition contained in Lib. A class defines the structure and behaviour for a collection of objects called its *instances*. The structure of a class is defined by its *attributes*:

```
3.          books : Set(Lib::Book);
4.          readers : Set(Lib::Reader);
```

The class Library defines two attributes named books and readers. The type of the attribute named books is Set(Lib::Book) meaning that legal values of the slot named books in instances of the class Library are sets of instances of the class named Book defined in the package named Lib.

```
5.          registerReader(r:Lib::Reader)
6.              self.readers := (self.readers->including(r))
7.          end
8.          shelveBook(b:Lib::Book)
9.              self.books := (self.books->including(b))
10.         end
```

Lines 5 - 10 define methods that can be used to add new readers and books to a library. The method registerReader has a single argument defined on line 5 called r. The type declaration for r defines that the method will be supplied with instances of the class named Reader which is defined in the package Lib. The body of the method is defined in line 6: the currently registered readers are updated with the new reader.

```
11.     borrow(bookName:String,readerName:String)
12.         if self.bookShelved(bookName) and
13.            self.registeredReader(readerName)
14.         then let book = self.findBook(bookName)
15.                    reader = self.findReader(readerName)
16.              in self.removeBook(bookName) []
17.                 reader.borrow(book)
18.              end
19.         else state.error("Library::borrow")
20.         endif
21.     end
```

Lines 11 - 21 show the definition of a method that defines how books are borrowed from a library. The body of the method provides examples of a conditional expression (lines 12 - 20) and a let expression that introduces local variables (lines 14 - 18), sequencing of execution (using the [] operator in line 16). The method body also includes a collection of method calls, for for example reader.borrow(book) on line 17 invokes the method named borrow on the object named reader and passes bookName as an argument.

Line 19 shows how MMT handles errors. The name state refers to an object that represents the MMT system. This object implements a number of system methods, one of which is used to report errors.

```
22.     return(bookName:String,readerName:String)
23.         if self.borrows(readerName,bookName)
24.         then let reader = self.findReader(readerName)
25.                    book = reader.findBook(bookName)
26.              in reader.return(book) []
27.                 self.return(book)
28.              end
29.         else state.error("Library::return")
30.         endif
```

```
31.     end
32.     return(b:Library::Book)
33.         self.books := (self.books->including(b))
34.     end
35.     removeBook(name:String)
36.         self.books := (self.books->reject(b | b.name = name))
37.     end
```

Lines 22 - 37 define methods used to return a book and remove a book that is currently borrowed.

The methods defined so far are all *imperative* since thay are used for their side effects on a library object. Methods may also be defined as *queries.* A query is used to produce a value in the context of a current object state; it does not cause any side effects. MML methods may be defined to be queries, imperative or a mixture of the two. The following is an example of a query:

```
38.     borrows(r:String,b:String):Boolean
39.         if self.registeredReader(r)
40.         then self.findReader(r).borrows(b)
41.         else false
42.         endif
43.     end
```

The query defined on lines 38 - 42 is supplied with a reader name and a book name and returns true when the reader currently borrows the book. Note that in addition to declaring the argument types, the method named borrows declares that its return value is of type Boolean.

```
44.     bookShelved(name:String):Boolean
45.         self.books->exists(b | b.name = name)
46.     end
47.     registeredReader(name:String):Boolean
48.         self.readers->exists(r | r.name = name)
49.     end
50.     findBook(name:String):Lib::Book
51.         self.books->select(b | b.name = name).selectElement()
52.     end
53.     findReader(name:String):Lib::Book
54.         self.readers->select(r | r.name = name).selectElement()
```

```
55.         end
56.     end
```

Lines 44 - 55 define a collection of queries. They provide examples of iteration expressions: exists and select. An exists expression is applied to a collection (in line 45 the set self.books) and returns true when one of the elements of the collection (referred to as b in the expression body on line 45) satisfies the boolean expression body. A select expression is applied to a collection and returns a subcollection whose elements all satisfy a given boolean expression. On line 51 a collection is created that consists of all books with the given name. A set has a method named selectElement that randomly returns an element from the set.

Books and readers in a library are both examples of indexed elements. An indexed element has a feature that is used to distinguish it from other elements of the same type. In the case of books and readers we will use a name as a distinguishing feature. Rather than duplicate a name attribute in classes for books and readers we will use inheritance. The following is a class of element that are indexed by name:

```
57.     class Indexed
58.         name : String;
59.         init(s:Seq(Instance)):Object
60.             self.name := (s->at(0)) []
61.             self
62.         end
63.         toString():String
64.             "<" + self.of.name + " " + self.name + ">"
65.         end
66.     end
```

Indexed provides examples of two standard MML features: object initialisation and objct display. When a new instance of a class is created the object is initialised by passing some initialisation values to the method named init. By default all classes have an empty init method that may be redefined. The class named Indexed defines an init method on lines 59 - 62. The method is passed a sequence of values and must return the receiver of the message (self). The method expects the first element of the initialisation values (at index 0) to be the name of the new object.

MMT displays objects by invoking their toString method (Java has a similar mechanism). By default, all objects have a toString method that cen be redefined. A general rule is that an object is displayed in the form <Type Data> where Type describes the type of the object and Data is some object-specific information about

the object. In the case of Indexed, the name of the class of the receiver is the type and the name is displayed as the data.

```
67.   class Book extends Lib::Indexed end
```

Line 67 shows how a book is defined as a sub-class of Indexed.

```
68.   class Reader extends Lib::Indexed
69.       books : Set(Lib::Book);
70.       borrow(b:Lib::Book)
71.           self.books := (self.books->including(b))
72.       end
73.       borrows(n:String):Boolean
74.           self.books->exists(b | b.name = n)
75.       end
76.       findBook(n:String):Lib::Book
77.           self.books->select(b | b.name = n).selectElement()
78.       end
79.       return(b:Lib::Book)
80.           self.books := (self.books->excluding(b))
81.       end
82.   end
```

Lines 68 - 82 complete the definition of the library classes. The class Reader is a sub-class of Indexed. Methods named borrow and return provide examples of including and excluding expressions which add and remove elements from sets.

MMT is an interactive environment. New definitions may be dynamically loaded and become part of the top-level name space. When developing a new package it is useful to define a test suite as a collection of package methods:

```
83.    test()
84.        let l = Lib::Library.new(Seq{})
85.            fred = Lib::Reader.new(Seq{"Fred"})
86.            barney = Lib::Reader.new(Seq{"Barney"})
87.            b1 = Lib::Book.new(Seq{"Book1"})
88.            b2 = Lib::Book.new(Seq{"Book2"})
89.        in Lib::printLibrary(l) []
90.           l.registerReader(fred) []
91.           l.registerReader(barney) []
92.           l.shelveBook(b1) []
```

```
93.             l.shelveBook(b2) []
94.             Lib::printLibrary(l) []
95.             l.borrow("Book1","Fred") []
96.             Lib::printLibrary(l) []
97.             l.borrow("Book2","Barney") []
98.             Lib::printLibrary(l) []
99.             l.return("Book1","Fred") []
100.            Lib::printLibrary(l)
101.        end
102.    end
```

Lines 83 - 102 define a package method called test. The method creates a number of local objects by instantiating the library classes and then performs a number of library operations using them. The printLibrary method is defined below:

```
103.    printLibrary(l:Lib::Library)
104.        ("Books = " + l.books.toString()).println() []
105.        "Readers:".println() []
106.        Lib::printReaders(l.readers) []
107.        "\n".print()
108.    end
109.    printReaders(s:Set(Lib::Reader))
110.        if not s->isEmpty
111.        then let r = s.selectElement()
112.             in Lib::printReader(r) []
113.                Lib::printReaders(s->excluding(r))
114.             end
115.        endif
116.    end
117.    printReader(r:Lib::Reader)
118.    ("  " + r.name).println() []
119.    ("  Books: " + r.books.toString()).println()
120.    end
121.end
```

The printing methods use the string operations called print and println to print to the standard output. The println operation is the same as print except it appends a newline (\n) character to the output.

The library package is loaded into MMT by clicking on the load button on the browser window, navigating to and selecting the file. Once loaded, the package named Lib can be referenced by name in the MMT base window:

```
MMT> Lib;
<Package Lib>
MMT>
```

The MMT base window provides a read-eval-print cycle that allows developers to interact with their models. For example the following transcript shows how the test method is invoked:

```
MMT> Lib::test();
Books = Set{}
Readers:
Books = Set{<Book Book1>,<Book Book2>}
Readers:
 Fred
 Books: Set{}
Barney
Books: Set{}


Books = Set{<Book Book2>}
Readers:
Fred
Books: Set{<Book Book1>}
Barney
Books: Set{}


 Books = Set{}
 Readers:
 Fred
 Books: Set{<Book Book1>}
 Barney
 Books: Set{<Book Book2>}

 Books = Set{<Book Book1>}
 Readers:
 Fred
 Books: Set{}
```

Barney
Books: Set{<Book Book2>}

CHAPTER 3

Data Values

MMT computes in terms of data values. The data values are: integers; booleans; strings; sets; sequences; arrays; functions and objects. Everything in MMT is a data value of one of these types. This chapter introduces the basic data value types.

Integers

MMT integers are all instances of the classifier named Integer. This classifier provides a collection of operations that may be performed on integers. Syntactic support for many of these operations is built in to the MMT parser so, for example, we write 10 * 2 rather than 10.mul(2). The following interface shows the main integer operations, the keyword **metaclass** declares that the MMT object named Integer is a data type and the keyword **extends** declares that all integers are MMT instances (the root of the MMT clas hierarchy).

1. **classifier** Integer **metaclass** DataType **extends** Instance
2. sqrt(other:Integer):Integer
3. add(other : Integer):Integer
4. sub(other : Integer):Integer
5. mul(other : Integer):Integer

6. div(other : Integer):Integer
7. greater(other : Integer):Boolean
8. greaterOrEql(other : Integer):Boolean
9. less(other : Integer):Boolean
10. lessOrEql(other : Integer):Boolean
11. to(upper : Integer):Seq(Integer)
12. isWhiteSpace():Boolean
13. max(other : Integer):Integer
14. equals(other : Instance):Boolean
15. **end**

Most integer operations should be self explanatory. The method named to generates a sequence of integers starting with the receiver and ending with the argument named upper.

The default value for an integer is 0.

Booleans

A boolean value is either true or false. These are builtin MMT constants with the following interface:

1. **classifier** Boolean **metaclass** DataType **extends** Instance
2. and(other : Boolean):Boolean
3. or(other : Boolean):Boolean
4. not(other : Boolean):Boolean
5. xor(other : Boolean):Boolean
6. equals(other:Instance):Boolean
7. **end**

The default boolean value is true.

Strings

A string is a sequence of characters. Strings are builtin MMT values with the following interface:

```
1.  classifier String metaclass DataType extends Instance
2.      print():Instance
3.      println():Instance
4.      less(s : String):Boolean
5.      greater(s : String):Boolean
6.      equals(s : String):Boolean
7.      add(s : String):String
8.      at(n : Integer):Integer
9.      hasPrefix(s : String):Boolean
10.     strip1(c : Integer):String
11.     stripPrefix(s : String):String
12.     stripLeadingWhiteSpace():String
13.     find(s : String):String
14.     nextName():String
15.     splitAt(char : Integer):Seq(String)
16.     repeat(n : Integer):String
17. end
```

The method print prints a string to the standard output. The method println prints and appends a newline character. Strings are compared using greater (>), less (<) and equals (=). Strings are concatenated using add (+). The characters in a string are indexed starting from 0, s->at(n) returns the character at index n. Th emethod hasPrefix tests whether the recsiever has the given prefix. The method strip1 removes the first occurrence of the given character. The method stripPrefix removes the given prefix. The method stripLeadingWhiteSpace removes leading spaces and newline characters and returns a new string. The method find takes a string and returns the longest suffix of the receiver that has the given string as a prefix. The method nextName returns the longest prefix of the receiver that consists of non-white space characters. The method splitAt returns a sequence of substrings formed by splitting the receiver at the given character. The method repeat returns a string that is the concatenation of the receiver with itself the given number of times.

The default string is empty “”.

A Calculator for Integer Expressions

The following example provides an example of integer operators in addition to some useful string and sequence operations:

```
1.  class Calc
2.      input:String;
3.      init(s:Seq(Instance)):Object
4.          self.input := (s->at(0)) []
5.          self
6.      end
```

A calculator is created with a numeric expression represented as a string. Examples of expressions are “10 + 2” and “10 + 2 > 37 - 3”. The evaluator for expressions is to be implemented as a recursive descent parser:

```
7.      exp():Integer
8.          let b = self.compare()
9.          in cond
10.             self.match('&') then
11.                 self.consumeChar() []
12.                 b and self.exp();
13.             self.match('|') then
14.                 self.consumeChar() []
15.                 b or self.exp();
16.             else b
17.         end
18.     end
19.   end
```

An expression is parsed and evaluated using the method named exp. Line 8 parses a comparison expression (involving operators ‘>’, ‘<‘ and ‘=’) which returns an integer or boolean b. Line 9 is the start of a conditional expression with multiple cases. The first case on line 10 tests to see of an ‘&’ operator follows the comparison expression. If so then the operator is consumed, the following boolean value is evaluated and combined with b using the infix boolean operator **and**. The second

case parses an or operator '|'. If no operators are found (line 16) then the evaluation is complete and the value b is returned.

An MMT character is surrounded by quote characters. Characters are represented ASCII integer codes and can therefore be compared using operators such as > and =. Control characters are prefixed by a backslash for example '\n' is the newline character.

An MMT string is enclosed in string quotes. A string s may be transformed into a sequence of characters using s->asSequence. The sequence may be processed as a sequence of integers (perhaps translating to upper case or adding and removing characters) and then translated back to a string using the method asString(). The following expression causes no change to the string s: s->asSequence.asString().

The following methods use the same techniques as exp to encode evaluators for various operators at different levels of precedence:

```
20.     compare():Integer
21.         let n = self.binary()
22.         in
23.             cond
24.                 self.match('>') then
25.                     self.consumeChar() []
26.                     n > self.compare();
27.                 self.match('<') then
28.                     self.consumeChar() []
29.                     n < self.compare();
30.                 self.match('=') then
31.                     self.consumeChar() []
32.                     n = self.compare();
33.                 else n
34.             end
35.         end
36.     end
37.     binary():Integer
38.         let n = self.unary()
39.         in
40.             cond
41.                 self.match('+') then
42.                     self.consumeChar() []
```

```
43.                     n + self.binary();
44.                 self.match('-') then
45.                     self.consumeChar() []
46.                     n - self.binary();
47.                 self.match('*') then
48.                     self.consumeChar() []
49.                     n * self.binary();
50.                  self.match('/') then
51.                     self.consumeChar() []
52.                     n / self.binary();
53.                 else n
54.             end
55.         end
56.     end
57.
58.     unary():Integer
59.         cond
60.             self.match('~') then
61.                 self.consumeChar() []
62.                  not self.unary();
63.             self.match('-') then
64.                 self.consumeChar() []
65.                  - self.unary();
66.              else self.number()
67.         end
68.     end
```

The recursive descent expression parser terminates at integer numbers. A number is a sequence of numeric characters or a general expression enclosed in parentheses. The number parser must arrange for a sequence of numeric characters to be translated into the equivalent integer.

```
69.     number():Integer
70.         if self.endOfInput()
71.         then state.error("Expecting a number.")
72.         else let n = self.peekInteger()
73.             in cond
74.                 self.isNumericChar(n) then
75.                     self.consumeChar() []
```

```
76.                     self.number(n);
77.                 self.match('(') then
78.                     self.consumeChar() []
79.                     let n = self.exp()
80.                     in self.consume(')') [] n
81.                     end;
82.                 else state.error("Expecting a numeric char.")
83.               end
84.           end
85.       endif
86.   end
```

The numeric expression parser calls a method with the same name on line 76. The call passes a single argument whereas the definition of number on line 69 has no arguments. This is an example of method *overloading* in MMT. A class may define a number of methods with the same name (either directly or via inheritance) providing that the methods all have a different number of arguments.

The method named number with arity 1 is used to parse a sequence of numeric characters and translate them into an integer. The method is recursive; it continually consumes integer characters and adds them to the running total n. Each time a new integer is added the running total must be multiplied by 10.

```
87.   number(n:Integer):Integer
88.       if self.endOfInput()
89.       then n
90.       else let m = self.peekInteger()
91.            in if self.isNumericChar(m)
92.            then self.consumeChar() []
93.                 self.number((n * 10) + m)
94.            else n
95.            endif
96.          end
97.      endif
98.  end
```

The parsing methods define in lines 1 - 98 use a number of auxilary methods to manipulate the underlying input stream. These methods are defined below.

```
99.   consume(c:Integer)
```

```
100.        if self.match(c)
101.        then self.consumeChar()
102.        else state.error("Expecting: " + c)
103.        endif
104.    end
105.
106.    match(c:Integer):Boolean
107.        self.stripLeadingWhiteSpace() []
108.        if self.endOfInput()
109.        then false
110.        else self.peek() = c
111.        endif
112.    end
```

The method named consume expects to find the character c as the next input element. If c is present then the character is consumed. The method named match returns true when the next input character is c.

```
113.    stripLeadingWhiteSpace()
114.        self.input := (self.input.stripLeadingWhiteSpace())
115.    end
```

The method named stripLeadingWhiteSpace on lines 113 - 115 continually consumes white space characters at the head of the input until the input is exhausted or a non-white space character is found. This uses a method defined by strings of the same name.

```
116.    endOfInput():Boolean
117.        self.input = ""
118.    end
```

The method endOfInput defined on lines 116 - 118 returns true when the sequence of input characters is exhausted.

```
119.    peek():Integer
120.        self.stripLeadingWhiteSpace() []
121.        self.input->asSequence->at(0)
122.    end
123.
124.    peekInteger():Integer
125.        self.peek() - '0'
```

```
126.    end
```

The ‘peeking’ methods defined on lines 119 - 126 are used to inspect the next input character without consuming it. Line 125 shows how numeric characters can be converted to integers by subtracting the ASCII character code for 0.

```
127.    consumeChar()
128.        self.input := (self.input->asSequence.tail.asString())
129.    end
```

A character is consumed by the method defined on lines 127 - 129. The input is translated to a sequence of characters. A sequence has slots for its head and tail. The head is removed and the resulting sequence is translated back to a string[1].

The class Calc is completed by defining a numeric character predicate:

```
130.    isNumericChar(n:Integer):Boolean
131.        n >= 0 and n < 10
132.    end
133. end
```

Collections

MMT implements two types of collections: sets and sequences. Values of these types are instances of the common super-class CollectionOfInstance that defines common operations.

```
1.  classifier CollectionOfInstance metaclass DataType extends Instance
2.      size():Integer
3.      isEmpty():Boolean
4.      asSet():Set(Instance)
5.      asSequence():Seq(Instance)
6.      exists(f : Closure):Boolean
```

1. The input would probably be represented as a sequence of integer character codes if we were doing this for real. Translating it back to a string gives the opportunity of showing how a string is transformed. Note also the parentheses around the expression after := in line 128, this is a quirk arising from the operator precedence of :=.

7. forAll(f : Closure):Boolean
8. **end**

The method size returns the number of elements in the collection. The method asSet translates the collection to a set. The method asSequence translates the collection to a sequence. The method exists returns true when the collection contains at least one element that satisfies the predicate f. The method forAll returns true when every element in the collection satisfies the predicate f.[2]

Sets

A set is an instance of the data type SetOfInstance. SetOfInstance is an instance of DataType and inherits from CollectionOfInstance:

1. **classifier** SetOfInstance **metaclass** DataType **extends** CollectionOfInstance
2. subset(s : Set(Instance)):Boolean
3. setUnion(s : Set(Instance)):Set(Instance)
4. setDifference(s : Set(Instance)):Set(Instance)
5. iterate(f : Closure, v : Instance):Instance
6. collect(f : Closure):Set(Instance)
7. reject(f : Closure):Set(Instance)
8. select(f : Closure):Set(Instance)
9. excluding(v : Instance):Set(Instance)
10. includes(v : Instance):Boolean
11. intersection(s : Set(Instance)):Set(Instance)
12. union(s : Set(Instance)):Set(Instance)
13. product(s : Set(Instance)):Set(Instance)

2. The language MML is OCL-like and as such provides specific syntax support for collection expressions using the arrow ->. For example: c->size, c->asSet and c->exists(x | x > 10). Each collection method that is defined by OCL is defined as an MML keyword. Since keywords cannot be used for method names it is therefore not currently possible to invoke s.size(). To get round this you can use << and >> around method names as in s.<<“size”>>(). In this context the << and >>protect the method name from being interpreted as an MML keyword.

```
14.     flatten():Set(Instance)
15.     power(Set(Set(Instance))
16.  end
```

Sequences

```
1.  classifier SeqOfInstance metaclass DataType extends CollectionOfInstance
2.      iterate(f : Function, v : Instance):Instance
3.      collect(f : Closure):Seq(Instance)
4.      reject(f : Closure):Seq(Instance)
5.      select(f : Closure):Seq(Instance)
6.      append(s : SeqOfInstance):Seq(Instance)
7.      at(i : Integer):Instance
8.      last():Instance
9.      separateWith(separator : String):String
10.     zip(s : Seq(Instance)):Instance
11.     flatten():Seq(Instance)
12.     remove(v : Instance):Seq(Instance)
13.     reverse():Seq(Instance)
14.     qsort():Seq(Instance)
15.     qsort(f : Closure):Seq(Instance)
16. end
```

Functions

MMT supports first class functions that can be created dynamically, stored in slots, passed as argumentsto methods (and functions) and returns as results. Functions provide a very flexible way of encoding behaviour and allow components to be parameterised with respect to a wide range of different behaviours.

MMT functions are instances of the class Closure which has the following interface:

```
1.  class Closure
2.      sourceCode : Exp ;
3.      self : Instance ;
4.      body : Array ;
5.      locals : Integer ;
6.      globals : Array ;
7.      args : Integer ;
8.      globalNames : Seq(String) ;
9.      argNames : Seq(String) ;
10.     apply(args : Seq(Instance)):Instance
11. end
```

A function is created by evaluating an expression of the form:

fun(<argList>) [:<type>] <exp> **end**

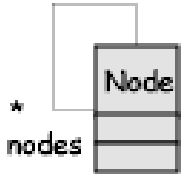
where <argList> is a comma separated sequence of arg declarations. An arg declaration is a name followed by an optional type of the form :<type>. The source code of the function body is the value of the sourceCode slot of the function (line 2). The source code is an expression object. MMT is a compiled system so changing the source code of a function will not have any effect (although it is possible to recompile a function this is not currently available).

Often a function is created in the context of a method in which case the special variable ‘self’ refers to the receiver of the message that caused the method to be invoked. A new function will capture the current value of self (line 3). This can be modified by updating this slot. The body of a function is an array of VM instructions (line 4). A function is performed by applying it to arguments, the number of arguments is the value of the slot args (line 7) and the names of the arguments are the valueof the slot argNames (line 9).

A function contains references to variables that are defined in a surrounding scope. These variabels are ‘global’ to the function; their names are held in the slot globalNames (line 8) and their corresponding values are in globals (line 6). The maximum number of local variables that the function requires (related to the size of the machine stack frame) is defined by locals (line 5).

A function is perfoemd by applying it to some arguments. The same effect can be achieved by invoking the apply method (line 10).

A Tree Manipulation Package



The following example shows how operations over a recursively defined structure can be parameterised with functions. The operations define the general mechanisms for updating and transforming the structure.

A tree node contains some data and a sequence of children nodes:

```
1.  package Trees
2.      class Node
3.          data : Instance;
4.          nodes : Seq(Trees::Node);
```

The data part of a node is supplied whtn the node is created and child nodes may be added using the addNode method. Line 11 provides an example of appending two sequences together:

```
5.          init(s:Seq(Instance)):Object
6.              self.data := (s->at(0)) []
7.              self
8.          end
9.
10.         addNode(n:Trees::Node)
11.             self.nodes := (self.nodes->append(Seq{n}))
12.         end
```

A useful tree operation involves applying an update to the data at each of the nodes in the tree. The mechanics of such a tree transformation is the same for every data update operation. MMT allows functions, or *closures*, as first class data values.

Functions may be passed as arguments to methods, returned from methods, held in collections and placed in object slots. Once created, a function is performed by supplying it with its arguments. The following method defined a tree transformation. The particular data update operation is supplied as the value of the argument f:

```
13.         transform(f:Closure)
14.             self.data := (f(self.data)) []
15.             self.nodes->collect(n | n.transform(f))
16.         end
```

The transformation method defined on lines 13 - 16 performed an update in place on all the data held in nodes. The operation therefore cannot change the shape of the tree or produce a value computed in terms of the tree data. A much more flexible operation can be defined in terms of functions. A node consists of a data part and a sequence of child nodes. The following method is supplied with a function that transforms the data part of each node and a binary function that combines the transformed data into new structures:

```
17.         fold(binFun:Closure,unFun:Closure):Instance
18.             self.nodes->iterate(n w = unFun(self.data) |
19.                 binFun(w,n.fold(binFun,unFun)))
20.         end
21.     end
```

The method named fold on lines 17 - 20 uses an iterate expression to transform and combine the data elements in a node and its children. The iterate expresion on lines 18 - 19 selects each child node n in turn. The expression initialises w by applying the unary function unFun to the data element. For each value of n the body of the iterate expresion is performed on line 19 and the resulting value is the new value of w for the next value of n. Each time round this loop the value of w is computed by applying binFun to the current value of w and the folded value of the child node n.

The class Node defined on lines 1 - 21 defines a simple tree structure with some general methods. The following method provides examples of how this could be used to represent a tree of integers and perform some operations on the tree:

```
22.     test()
23.         let n1 = Trees::Node.new(Seq{1})
24.             n2 = Trees::Node.new(Seq{2})
25.             n3 = Trees::Node.new(Seq{3})
26.             n4 = Trees::Node.new(Seq{4})
```

```
27.          n5 = Trees::Node.new(Seq{5})
28.          n6 = Trees::Node.new(Seq{6})
29.         n7 = Trees::Node.new(Seq{7})
30.         n8 = Trees::Node.new(Seq{8})
31.         id = fun(x) x end
32.     in  n1.addNode(n2) []
33.         n1.addNode(n3) []
34.         n2.addNode(n4) []
35.         n2.addNode(n5) []
36.         n3.addNode(n6) []
37.         n3.addNode(n7) []
38.         n7.addNode(n8) []
```

Lines 22 - 38 create nodes n1 to n8 containing integers. The function named id performs the identity operation. The addNode method is used to link up the nodes into a tree structure.

```
39.         n1.transform(fun(x) x + 1 end) []
```

The first example of a tree operation is shown on line 39 where every node is updated by adding 1 to its data element.

```
40.         n1.fold(fun(x,y) x + y end,id).toString().println() []
```

Line 40 folds the tree by adding up every element and then printing the result to the standard output.

```
41.         n1.fold(fun(x,y) x * y end,id).toString().println() []
```

Line 41 folds the tree by multiplying all the values together and then printing the result.

```
42.         n1.fold(
43.             fun(s1,s2)
44.                 s1->append(s2)
45.             end,
46.             fun(x)
47.                 Seq{x}
48.             end).toString().println() []
```

Lines 42 - 48 define a fold that transforms a tree by flattening the structure and returning a sequence of integers that is then printed to the standard output. This is an example showing how the folding mechanism can be used to transform from one structure to another. The following example shows how a tree can be transformed into another tree of the same shape but where the data has been changed. The example involves two fold transformations placed in sequence. The first transformation copies the tree structure adding 10 to each data element. The second transformation checks whether all elements in the tree are greater than 11:

```
49.              n1.fold(
50.                  fun(node,child)
51.                      node.addNode(child) []
52.                      node
53.                  end,
54.                  fun(x)
55.                      Trees::Node.new(Seq{x+10})
56.                  end).fold(
57.                      fun(x,y)
58.                          x and y
59.                      end,
60.                      fun(x)
61.                          x > 11
62.                      end).toString().println()
63.          end
64.      end
65.
66. end
```

Free Variables

The source code of a function is an instance of the class Exp. All features of MMT are represented as instances of MMT classes and MML is no exception. Access to expressions is important because MMT can be used to construct models of systems or prototype implementations and then transformations can be applied to both the structure and behaviour of the system in order to realise it as a concrete implementation.

This section defines a package FreeVars that implements a method to calculate the free variables of a function. The package is not particularly interesting in itself (although it is used as part of the compiler for MMT) however it uses all the classes representing MML source code and in particular their slots.

```
1.  package FreeVars
2.      freeVars(f:Closure):Set(String)
3.          let  e = f.sourceCode
4.               a = f.argNames
5.          in FreeVars::free(e).setDifference(a)
6.          end
7.      end
```

Lines 1 - 7 introduce the package FreeVars. The method freeVars is applied to a closure and produces a set of free variable names defined in the body of the closure.

```
8.  free(e:Exp):Set(String)
9.      cond
10.         e.isKindOf(Add) then
11.             // e1 + e2
12.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
```

The method free (line 8) is applied to an expression and produces a set of free variable names defined in the expression. Thr method proceeds by case analyisis on the type of the expression e. Each case, tests the type of e and then calculates the free variables in terms of the structure of the expression. In each case the calculation involves referencing all the structural slots of the expression object.

Lines 10 - 12 show how the free variables of an addition expression are constructed. An addition expression consists of a left and a right sub-expression. The free varibles of the addition expression are the union of the free varibles of the two sub-expressions.

```
13.         e.isKindOf(And) then
14.             // e1 and e2
15.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
16.         e.isKindOf(Append) then
17.             // e1->append(e2)
18.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
19.         e.isKindOf(Apply) then
```

```
20.             // e(e1,e2,...,en)
21.             e.operands->iterate(x S = FreeVars::free(e.operator) |
22.                 S->union(FreeVars::free(x)));
23.         e.isKindOf(AsSequence) then
24.             // e->asSequence
25.             FreeVars::free(e.exp);
26.         e.isKindOf(AsSet) then
27.             // e->asSet
28.             FreeVars::free(e.exp);
29.         e.isKindOf(At) then
30.             //e1->at(e2)
31.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
32.         e.isKindOf(AttributeDef) then
33.             // n : t
34.             Set{};
```

Lines 13 - 34 define how to construct the free variables of and expressions to attribute definitions in class definitions. A class definition is as follows:

```
35.         e.isKindOf(ClassDef) then
36.             // class name metaclass meta extends supers atts methods inv end
37.             FreeVars::free(e.meta)->union(
38.                 e.supers->iterate(x S = Set{} |
39.                     S->union(FreeVars::free(x)))->union(
40.                         e.attributes->iterate(a S = Set{} |
41.                             S->union(FreeVars::free(a)))->union(
42.                                 e.methods->iterate(m S = Set{} |
43.                                     S->union(FreeVars::free(m)))->union(
44.                                         e.invariants->iterate(i S = Set{} |
45.                                             S->union(FreeVars::free(i)))))));
```

A class definition consists of a metaclass (an expression), a sequence of super classes (expressions), some attributes (attributes definitions), some methods (method definitions) and invariants (constraint definitions).

```
46.         e.isKindOf(Collect) then
47.             // e1->collect(v | e2)
48.             FreeVars::free(e.coll)->union(FreeVars::free(e.body)->excluding(e.var));
49.         e.isKindOf(Div) then
```

```
50.             // e1 / e2
51.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
52.         e.isKindOf(Eql) then
53.             // e1 = e2
54.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
55.         e.isKindOf(Excluding) then
56.             // e1->excluding(e2)
57.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
58.         e.isKindOf(Exists) then
59.             // e1->exists(v | e2)
60.             FreeVars::free(e.coll)->union(FreeVars::free(e.body)->excluding(e.var));
61.         e.isKindOf(FieldRef) then
62.             // e1.n
63.             FreeVars::free(e.obj);
64.         e.isKindOf(First) then
65.             // e1->first
66.             FreeVars::free(e.exp);
67.         e.isKindOf(Follows) then
68.             // e1 [] e2
69.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
70.          e.isKindOf(ForAll) then
71.             // e1->forAll(v | e2)
72.             FreeVars::free(e.coll)->union(FreeVars::free(e.body)->excluding(e.var));
73.        e.isKindOf(Function) then
74.             // fun(args) body end
75.             FreeVars::free(e.body).setDifference(e.args->asSet);
76.        e.isKindOf(Greater) then
77.             // e1 > e2
78.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
79.        e.isKindOf(GreaterOrEql) then
80.             // e1 >= e2
81.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
82.        e.isKindOf(Id) then
83.             // name
84.             Set{e.name};
85.        e.isKindOf(If) then
86.             // if test then consequent else alternative endif
```

```
87.             FreeVars::free(e.test)->union(
88.                 FreeVars::free(e.consequent)->union(
89.                     FreeVars::free(e.alternative)));
90.         e.isKindOf(Implies) then
91.             // e1 implies e2
92.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
93.         e.isKindOf(Includes) then
94.             // e1->includes(e2)
95.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
96.         e.isKindOf(Including) then
97.             // e1->including(e2)
98.             FreeVars::free(e.left)->union(FreeVars::free(e.right));
99.         e.isKindOf(Intersection) then
100.            // e1->intersection(e2)
101.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
102.        e.isKindOf(IsEmpty) then
103.            // e->isEmpty
104.            FreeVars::free(e.exp);
105.        e.isKindOf(Iterate) then
106.            // coll->iterate(var local = value | body)
107.            FreeVars::free(e.coll)->union(
108.                FreeVars::free(e.value)->union(
109.                    FreeVars::free(e.body)->excluding(e.local)->excluding(e.var)));
110.        e.isKindOf(Last) then
111.            // e1->last
112.            FreeVars::free(e.exp);
113.        e.isKindOf(Less) then
114.            // e1 < e2
115.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
116.        e.isKindOf(LessOrEql) then
117.            // e1 <= e2
118.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
119.        e.isKindOf(Let) then
120.            // let v = e1 in e2 end
121.            FreeVars::free(e.value)->union(
122.                FreeVars::free(e.body)->excluding(e.local));
123.        e.isKindOf(Literal) then
124.            // an integer, string or boolean
```

```
125.            Set{};
126.        e.isKindOf(Mul) then
127.            // e1 * e2
128.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
129.        e.isKindOf(Neg) then
130.            // - e
131.            FreeVars::free(e.exp);
132.        e.isKindOf(Not) then
133.            // not e
134.            FreeVars::free(e.exp);
135.        e.isKindOf(NotEql) then
136.            // e1 <> e2
137.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
138.        e.isKindOf(Or) then
139.            // e1 or e2
140.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
141.        e.isKindOf(PairExp) then
142.            // Seq{e1 | e2}
143.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
144.        e.isKindOf(Prepend) then
145.            // e1->prepend(e2)
146.            FreeVars::free(e.left)->union(FreeVars::free(e.right));
147.        e.isKindOf(Reject) then
148.            // e1->reject(v | e2)
149.            FreeVars::free(e.coll)->union(FreeVars::free(e.body)->excluding(e.var));
150.        e.isKindOf(Select) then
151.            // e1->select(v | e2)
152.            FreeVars::free(e.coll)->union(FreeVars::free(e.body)->excluding(e.var));
153.        e.isKindOf(Send) then
154.            // e.m(e1,...,en)
155.            e.args->iterate(a S = FreeVars::free(e.obj) | S->union(FreeVars::free(a)));
156.        e.isKindOf(SeqExp) then
157.            // Seq{e1,e2,...,en}
158.            e.elements->iterate(x S = Set{} | S->union(FreeVars::free(x)));
159.        e.isKindOf(SetExp) then
160.            // Set{e1,e2,...,en}
161.            e.elements->iterate(x S = Set{} | S->union(FreeVars::free(x)));
```

```
        e.isKindOf(Size) then
            // e->size
            FreeVars::free(e.exp);
        e.isKindOf(Sub) then
            // e1 - e2
            FreeVars::free(e.left)->union(FreeVars::free(e.right));
        e.isKindOf(SubSequence) then
            // e->subsequence(e2,e2)
            FreeVars::free(e.coll)->union(
                FreeVars::free(e.lower)->union(
                    FreeVars::free(e.upper)));
        e.isKindOf(SymmetricDifference) then
            // e1->symmetricDifference(e2)
            FreeVars::free(e.left)->union(FreeVars::free(e.right));
        e.isKindOf(Union) then
            // e1->union(e2)
            FreeVars::free(e.left)->union(FreeVars::free(e.right));
        e.isKindOf(Update) then
            // e1.n := e2
            FreeVars::free(e.obj)->union(
                FreeVars::free(e.value));
         e.isKindOf(Xor) then
            // e1 xor e2
             FreeVars::free(e.left)->union(FreeVars::free(e.right));
         else state.error("FreeVars::free: unknown expression " + e.toString())
     end
end
```

Instances

Eveything in MMT is an instance of some classifier. The class called Instance is the root of the class hierarchy and therefore all classes inherit the features defined by this class:

```
1. class Instance
2.     of : Classifier
3.     isKindOf(c : Classifier):Boolean
```

```
4.      equals(other : Instance):Boolean
5.      init():Instance
6.      send(message : String, args : Seq(Instance)):Instance
7.  end
```

Every instance has a slot named of that contains its classifier. The method isKindOf is used to test whether an instance against any given classifier. Instances can be compared using the method named equals. All instances can be initialised using the method named init (this is distinct from the method with the same name defined by Object). The method named send is used to send a message to an instance.

Objects

```
8.  class Object extends Instance
9.      slots : Set(Slot)
10.     toString():String
11.     init(s : Seq(Instance)):Object
12.     initSlots():Object
13.     slotValue(name : String):Instance
14.     hasSlot(name : String):Boolean
15.     setSlot(name : String, value : Instance)
16.     copy():Object
17. end
```

DataBase Queries

The following example shows how objects, collections and functions can be combined to implement a simple model of relational databases. The example also shows how builtin operators such as + are overloaded for user defined classes.

A relational database table consists of columns and rows. Each column has a name and a type. Each row contains a value of the specified type for each column. SQL is

used to perform queries and updates on tables. Suppose that we have the following table called ageTable:

name : String	**age : Integer**
Fred	41
Wilma	31
Barney	38
Bam Bam	3
Bart	9
Homer	38
Pebbles	3

The following SQL query:

```
SELECT name,age FROM ageTable WHERE age > 30
```

produces the following table:

name : String	**age : Integer**
Fred	41
Wilma	31
Barney	38
Homer	38

Tables may be joined on common fields. The following table is called addressTable:

name : String	**address : String**
Fred	Bedrock
Wilma	Bedrock
Barney	Bedrock
Bam Bam	Bedrock
Bart	Springfield

name : String	address : String
Homer	Springfield
Pebbles	Bedrock

By joining ageTable and addressTable we get:

name : String	age : Integer	address : String
Fred	41	Bedrock
Wilma	31	Bedrock
Barney	38	Bedrock
Bam Bam	3	Bedrock
Bart	9	Springfield
Homer	38	Springfield
Pebbles	3	Bedrock

SQL allows rows to be deleted:

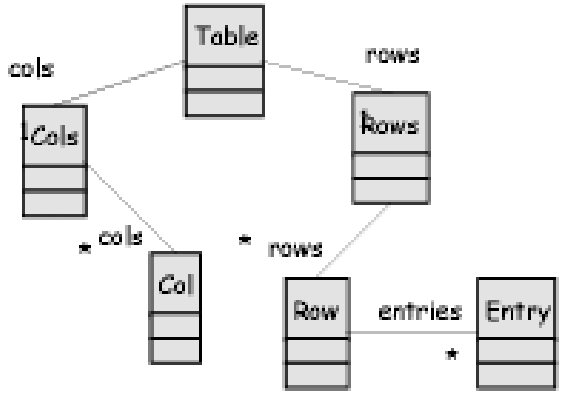
```
DELETE FROM ageTable WHERE age < 10
```

which updates the table by removing all rows that satisfy the condition. SQL allows new rows to be added:

```
INSERT IN ageTable (name,age) (Betty,36)
```

Finally, SQL allows fields to be updated:

```
UPDATE IN ageTable married = false WHERE age < 16
```



The following package defines a collection of classes that represent database tables. SQL queries are represented as functions that range over instances of the classes.

```
1.  package DataBase
2.      class Table
3.          rows : DataBase::Rows;
4.          cols : DataBase::Cols;
5.          init(s:Seq(Instance)):Object
6.              let  rows = s->at(0)
7.                   cols = s->at(1)
8.              in   self.rows := rows []
9.                   self.cols := cols []
10.                  self
11.             end
12.         end
```

Lines 2 - 12 introduce a class that represents database tables. Each table consists of a rows and columns. Rather than represent the values of the rows and cols attributes of the class as sets of objects, we represent them as objects that *contain* sets of objects (see below). This allows tables to *delegate* messages to all the rows and columns.

```
13.         toString():String
14.             self.cols.toString() + "\n" +
15.             "-".repeat(14).repeat(self.cols.names()->size) + "\n" +
```

```
16.             self.rows.toString(self.cols.names()) + "\n"
17.         end
```

Lines 13 - 17 define a toString method for database tables. A table is to be printed out in tabular format: the column headings are printed followed by a line of '-' characters then the rows are printed out. The rows are supplied with a sequence of column names to ensure that the order in which the values are printed is consistent.

```
18.         insert(values:Seq(Instance))
19.             self.rows.insert(self.colNames(),values)
20.         end
21.         update(pred:Closure,updater:Closure):DataBase::Table
22.             self.rows.update(pred,updater) [] self
23.         end
24.         delete(pred:Closure):DataBase::Table
25.             self.rows.delete(pred) [] self
26.         end
```

Lines 18 - 26 define methods for inserting a sequence of values as a row, updating every row that satisfies a condition and deleting every row that satisfies a condition. The method named update takes two functions as arguments. The first function named pred expects a single row argument and returns either true or false. The second function expects a single row argument and performs an update operation. The method named delete expects a single function argument. The function expects a single row argument and returns true when the row is to be deleted from the table.

```
27.         colNames():Seq(String)
28.             self.cols.names()
29.         end
```

Lines 27 - 29 define an auxiliary table operation that returns a sequence of column names. The table delegates responsibility of computing the names to the cols object.

```
30.         sel(names:Set(String),pred:Closure):DataBase::Table
31.             DataBase::Table.new(Seq{self.rows.sel(names,pred),self.cols.sel(names)})
32.         end
```

Lines 30 - 32 define the selection operator that filters a table with respect to a condition. The method named sel is supplied with a set of column names and a function. The result of the selection is a new table with the given column names. The

new table will contain copies of the rows from the receiver that satisfy the predicate and have been trimmed to just have the given column names.

```
33.         add(table:DataBase::Table):DataBase::Table
34.             DataBase::Table.new(Seq{self.rows + table.rows,self.cols + table.cols})
35.         end
36.     end
```

Lines 33 - 35 define the join operator. The method is named add because MMT allows classes to overload the + operator. This conveniently means that given two tables, for example ageTable and addressTable, their join can be expressed by adding them together: ageTable + addressTable.

A collection of columns is defined by the class Cols:

```
37.     class Cols
38.         cols : Seq(DataBase::Col);
39.         init(s:Seq(Instance)):Object
40.             self.cols := (s->at(0)) []
41.             self
42.         end
```

The columns are represented as a sequence so that they have a deterministic ordering. The display method for a collection of columns turns each column into a string and then separates the resulting sequence with '|' characters. Note that the collect expression is supplied with a sequence but returns a set and therefore must be translated into a sequence using asSequence.[3]

```
43.         toString():String
44.             self.cols->collect(col | col.toString())->asSequence.separateWith(" |")
45.         end
```

The names of the columns are calculated by a method:

```
46.         names():Seq(String)
47.             self.cols->collect(col | col.name)->asSequence
48.         end
```

3. This is an undesirable feature of MMT collection expressions - they all return sets. This should be fixed in a future release.

it is important that the names are returned in the same order each time[4].

```
49.         binds(name:String):Boolean
50.             self.names()->includes(name)
51.         end
52.          get(name:String):Classifier
53.              self.cols->select(col | col.name = name).selectElement().type
54.         end
```

Lines 49 - 54 define methods that check whether there is a column with a given name and that returns the type of a given column name.

Lines 55 - 58 define selection for columns. Given a set of names, selection returns a new set of columns with the given names.

```
55.         sel(names:Set(String)):DataBase::Cols
56.             DataBase.Cols.new(Seq{self.cols->select(col |
57.                 names->includes(col.name))->asSequence})
58.         end
```

Two sets of columns are joined together to produce a single set of columns cntaining the union of the two sets. The resulting set should contain a single entry for each column with a given name (we assume that names occurring in both of the operands of the join have the same type).

```
59.         add(cols:DataBase::Cols):DataBase::Cols
60.             DataBase::Cols.new(Seq{self.cols->append(cols.cols->reject(col |
61.                 self.names()->includes(col.name))->asSequence)})
62.         end
63.     end
```

A single column has a name and a type. The type of the column will be one of Integer, String or Boolean each of which is an object of type DataType:

```
64.     class Col
65.         name : String;
66.         type : DataType;
67.         init(s:Seq(Instance)):Object
```

4. Although collection expressions currently return sets, the underlying representation of a set is ordered and will produce the same sequence each time.

```
68.             self.name := (s->at(0)) []
69.             self.type := (s->at(1)) []
70.             self
71.         end
```

A column is displayed by concatenating its name and type and ensuring that the resulting string fits into a standard width. The string manipulation is performed using two auxiliary methods padTo and trunc that are defined later in the package DataBase:

```
72.         toString():String
73.             DataBase::trunc(
74.                 DataBase::padTo(
75.                     " " + self.name + ":" + self.type.name,12),12)
76.         end
77.         end
```

The rows of a database are represented as an instance of the class Rows:

```
78.     class Rows
79.         rows : Set(DataBase::Row);
80.         init(s:Seq(Instance)):Object
81.             self.rows := (s->at(0)) []
82.             self
83.         end
```

The rows are not ordered and therefore the attribute named rows in line 79 is a set. A collection of rows are displayed by displaying each row in turn and separating the rows with a newline character:

```
84.         toString(names:Seq(String)):String
85.             self.rows->collect(row |
86.                 row.toString(names))->asSequence.separateWith("\n")
87.         end
```

A new row is added using the method insert:

```
88.         insert(names:Seq(String),values:Seq(Instance))
89.             let pairs = names.zip(values)
90.                 entries = pairs->collect(initArgs | DataBase::Entry.new(initArgs))
91.                 row = DataBase::Row.new(Seq{entries})
```

```
92.             in self.addRow(row)
93.             end
94.         end
95.         addRow(r:DataBase::Row)
96.             self.rows := (self.rows->including(r))
97.         end
```

The sequence method named zip is used in line 89 to translate a pair of sequences to a single sequence of pairs. Each pair is then the initialisation arguments in the creation of a new row entry in line 90.

When rows are selected by truncating the rows to the appropriate named columns and then filtering the rows that satisfy the given predicate:

```
98.         sel(names:Set(String),pred:Closure):DataBase::Row
99.             DataBase::Rows.new(Seq{self.rows->iterate(row R=Set{} |
100.                 let row' = row.sel(names)
101.                 in  if pred(row')
102.                     then R->including(row')
103.                     else R
104.                     endif
105.                 end)})
106.         end
```

Each row is exminated in turn by the iterate expression on line 99. The set R is a new collection of filtered and truncated rows. Each row is truncated by selecting the named entries on line 100. If the predicate is satisfied on line 101 then the truncated row is added to the set R (line 102) otherwise R is unchanged for the next iteration (line 103).

Two collections of rows are joined together by examining each pair of rows in turn:

```
107.         add(rows:DataBase::Rows):DataBase::Rows
108.             DataBase::Rows.new(Seq{self.rows->iterate(row R = Set{} |
109.                 rows.rows->iterate(row' R = R |
110.                     if row.canJoin(row')
111.                     then R->including(row + row')
112.                     else R
113.                     endif))})
114.         end
```

The predicate named canJoin on line 110 tests whether the two rows have the same values for entries with the same name. If this is the case then the two rows are joined (line 111) otherwise the combination of the two rows is illegal and the running collection of joined rows R is left unchanged for the next iteration (line 112).

Rows are updated and deleted using the following methods:

```
115.        update(pred:Closure,updater:Closure)
116.            self.rows->collect(row |
117.                if pred(row)
118.                then updater(row)
119.                endif)
120.        end
121.        delete(pred:Closure)
122.            self.rows := (self.rows->reject(row | pred(row)))
123.        end
124.    end
```

A row is a set of entries:

```
125.  class Row
126.      entries : Set(DataBase::Entry);
127.      init(s:Seq(Instance)):Object
128.          self.entries := (s->at(0)) []
129.          self
130.      end
131.      toString(names:Seq(String)):String
132.          names->collect(name |
133.              let display = self.ref(name).toString()
134.              in DataBase::trunc(DataBase::padTo(" "+display,12),12)
135.              end)->asSequence.separateWith(" |")
136.      end
```

The toString method on lines 131 - 136 displays the entry values of a row in fields 12 characters wide seperated with '|' characters.

A row can be thought of as a set of values indxed by names. The following methods allow the values to be referenced and updated by name:

```
137.      ref(name:String):Instance
138.          if self.binds(name)
```

```
139.            then self.get(name)
140.            else state.error("Row::ref: no field named " + name)
141.            endif
142.        end
143.        update(name:String,value:Instance)
144.            if self.binds(name)
145.            then self.set(name,value)
146.            else state.error("Row::update: no field named " + name)
147.            endif
148.        end
149.        set(name:String,value:Instance)
150.            self.entries->select(entry |
151.                entry.name = name).selectElement().value := value
152.            end
153.            binds(name:String):Boolean
154.                self.entries->exists(entry | entry.name = name)
155.            end
156.            get(name:String):Instance
157.                self.entries->select(entry | entry.name = name).selectElement().value
158.            end
```

The select operator for a row produces a new row containing just the selected names. Each of the entries in the new row are copies. The means that updates to the new row will not affect the original row and vice versa.

```
159.        sel(names:Set(String)):DataBase.Row
160.            DataBase::Row.new(Seq{
161.                self.entries->select(entry |
162.                    names->includes(entry.name))->collect(entry |
163.                        entry.copy())})
164.        end
```

The join of two rows is defined by the method named add. The predicate canJoin is used to test whether two rows are compatible:

```
165.        canJoin(row:DataBase::Row):Boolean
166.            self.names()->forAll(name |
167.                if row.binds(name)
168.                then self.get(name) = row.get(name)
```

```
                    else true
                    endif)
            end
            add(row:DataBase::Row):DataBase::Row
                DataBase::Row.new(Seq{self.entries->union(row.entries->reject(entry |
                    self.names()->includes(entry.name)))})
            end
    end
```

The class Entry just associates names and values:

```
    class Entry
        name : String;
        value : Instance;
        init(s:Seq(Instance)):Object
            self.name := (s->at(0)) []
            self.value := (s->at(1)) []
            self
        end
    end
```

Tables are displayed in a uniform size column format. The package DataBase provides two auxiliary methods to support this:

```
    trunc(s:String,len:Integer):String
        s->asSequence.zip((0).to(s->size - 1))->iterate(pair S = Seq{} |
            if pair->at(1) < len
            then S->append(Seq{pair->at(0)})
            else S
            endif).asString()
    end
    padTo(s:String,len:Integer):String
        s + (" ".repeat(len-(s->size)))
    end
```

The following test suite is used for databases:

```
    test()
        let col1 = DataBase::Col.new(Seq{"name",String})
            col2 = DataBase::Col.new(Seq{"age",Integer})
```

```
199.            col3 = DataBase::Col.new(Seq{"address",String})
200.            col4 = DataBase::Col.new(Seq{"married",Boolean})
201.            cols1 = DataBase::Cols.new(Seq{Seq{col1,col2}})
202.            cols2 = DataBase::Cols.new(Seq{Seq{col1,col3}})
203.            cols3 = DataBase::Cols.new(Seq{Seq{col1,col4}})
204.            rows1 = DataBase::Rows.new(Seq{Set{}})
205.            rows2 = DataBase::Rows.new(Seq{Set{}})
206.            rows3 = DataBase::Rows.new(Seq{Set{}})
207.            ageTable = DataBase::Table.new(Seq{rows1,cols1})
208.            addressTable = DataBase::Table.new(Seq{rows2,cols2})
209.            marriedTable = DataBase::Table.new(Seq{rows3,cols3})
210.        in ageTable.insert(Seq{"Fred",41}) []
211.       ageTable.insert(Seq{"Wilma",31}) []
212.       ageTable.insert(Seq{"Barney",38}) []
213.       ageTable.insert(Seq{"Bam Bam",3}) []
214.       ageTable.insert(Seq{"Bart",9}) []
215.       ageTable.insert(Seq{"Homer",38}) []
216.       ageTable.insert(Seq{"Pebbles",3}) []
217.       addressTable.insert(Seq{"Fred","Bedrock"}) []
218.       addressTable.insert(Seq{"Wilma","BedRock"}) []
219.       addressTable.insert(Seq{"Barney","Bedrock"}) []
220.       addressTable.insert(Seq{"Bam Bam","Bedrock"}) []
221.       addressTable.insert(Seq{"Bart","Springfield"}) []
222.       addressTable.insert(Seq{"Homer","Springfield"}) []
223.       addressTable.insert(Seq{"Pebbles","Bedrock"}) []
224.       marriedTable.insert(Seq{"Fred",true}) []
225.       marriedTable.insert(Seq{"Wilma",true}) []
226.       marriedTable.insert(Seq{"Barney",true}) []
227.       marriedTable.insert(Seq{"Bam Bam",false}) []
228.       marriedTable.insert(Seq{"Bart",false}) []
229.       marriedTable.insert(Seq{"Homer",true}) []
230.       marriedTable.insert(Seq{"Pebbles",false}) []
```

The main database table is the join of the age, address and married tables:

```
231.       let table = ageTable + addressTable + marriedTable
```

The following lines show how tables are displayed:

232. in ageTable.toString().println() []

```
name:String | age:Integer
---------------------------
 Fred         | 4
 Wilma        | 31
 Barney       | 38
 Bam Bam      | 3
 Bart         | 9
 Homer        | 38
 Pebbles      | 3
```

233. addressTable.toString().println() []

```
name:String | address:Str
----------------------------
 Fred         | Bedrock
 Wilma        | BedRock
 Barney       | Bedrock
 Bam Bam      | Bedrock
 Bart         | Springfield
 Homer        | Springfield
 Pebbles      | Bedrock
```

234. marriedTable.toString().println() []

```
name:String | married:Boo
----------------------------
 Fred         | true
 Wilma        | true
 Barney       | true
 Bam Bam      | false
 Bart         | false
```

```
 Homer       | true
 Pebbles     | false
```

```
235.            (ageTable + addressTable).toString().println() []
```

```
 name:String | age:Integer | address:Str
-----------------------------------------
 Fred        | 41          | Bedrock
 Wilma       | 31          | BedRock
 Barney      | 38          | Bedrock
 Bam Bam     | 3           | Bedrock
 Bart        | 9           | Springfield
 Homer       | 38          | Springfield
 Pebbles     | 3           | Bedrock
```

```
236.        table.toString().println() []
```

```
 name:String | age:Integer | address:Str | married:Boo
-------------------------------------------------------
 Fred        | 41          | Bedrock     | true
 Wilma       | 31          | BedRock     | true
 Barney      | 38          | Bedrock     | true
 Bam Bam     | 3           | Bedrock     | false
 Bart        | 9           | Springfield | false
 Homer       | 38          | Springfield | true
 Pebbles     | 3           | Bedrock     | false
```

```
237.            table.sel(
238.                Set{"name","age","married"},
239.                fun(r)
240.                    r.ref("age") > 35
241.                end).toString().println() []
```

```
 name:String | age:Integer | married:Boo
```

```
----------------------------------------
 Fred         | 41           | true
 Barney       | 38           | true
 Homer        | 38           | true
```

```
242.            table.sel(
243.                Set{"name","address"},
244.                fun(r)
245.                    r.ref("address") = "Springfield"
246.                end).sel(
247.                    Set{"name"},
248.                    fun(r)
249.                        true
250.                    end).toString().println() []
```

```
name:String
--------------
 Bart
 Homer
```

```
251.        (ageTable + marriedTable).update(
252.            fun(r)
253.                r.get("name") = "Bart"
254.            end,
255.            fun(r)
256.                r.update("married",true)
257.            end).toString().println() []
```

```
name:String | age:Integer | married:Boo
----------------------------------------
 Fred         | 41           | true
 Wilma        | 31           | true
 Barney       | 38           | true
 Bam Bam      | 3            | false
 Bart         | 9            | true
 Homer        | 38           | true
```

```
 Pebbles     | 3           | false
```

```
258.        (ageTable + addressTable).delete(
259.            fun(r)
260.                r.get("age") > 20
261.            end).toString().println()
```

```
name:String | age:Integer | address:Str
-----------------------------------------
 Bam Bam     | 3           | Bedrock
 Bart        | 9           | Springfield
 Pebbles     | 3           | Bedrock
```

State and Debugging

The class State describes the MMT Virtual Machine. The distinguished variable ‘state’ is always bound to the currently executing instance of the class State. You should not use ‘state’ as a variable name. The methods of State are machine utilities and can be used to interact with the operating system and to debug MMT.

CHAPTER 4

Name Spaces

A name space is a container of named values. The values in a name space can be referred to by name with respect to the containing name space. MMT provides a special syntax construct for name spaces: the infix '::' operator. Objects that implement name spaces are instances of the MMT class NameSpaces::NameSpace.

The most common forms of name spaces in MMT are classes and packages. Classes provide name spaces for their attributes, methods and invariants. Packages provide name spaces for the definitions that they contain. For example, MMT provides a package called Classes that contains definitions for class-like things. Amongst these are the classes named Class and Classifier. These classes can be referenced as Classes::Class and Classes::Classifier.

Classes define attributes named parents and methods. Given a class C the name space lookups C::attributes and C::methods produce the collections of attributes and methods defined by C. Returning to the Classes package, Classes::Class::attributes produces the attributes defined by the class Class and Classes::Classifier::methods produces the methods defined by the class Classifier.

Name spaces may name different categories of things. For example, a class names attributes and methods. In general, the names of different categories of things named in a name space may overlap. For example, there could be an attribute named x and a method named x in the same class. Which one will be returned by

C::x? In this case the name space mechanism via '::' is no sufficient to support multiple categories and C should provide a method for each category. For example classes provide findAttribute and getMethod. Where a name is known not to be shared between categories '::' can be used safely.

```
1. class NameSpace
2.     get(name : String):Set(Instance)
3.     getOne(name : String):Instance
4.     defines(name : String):Boolean
5. end
```

Lines 1 - 4 define the NameSpace interface. The method get (line 2) is used to lookup a name in a name space and returns a set of values with the given name. The method getOne (line 3) assumes that the name is defined in the name space and returns exactly on value for the name. If the name has multiple definitions then get-One must choose one: the default behaviour for getOne is to choose at random.

The predicate defines (line 4) is used to check whether a name space provides a definition for a given name.

The syntax construct X::y is translated directly to X.getOne("y").

CHAPTER 5

Classes

Introduction

Like most object-oriented systems MMT is class-based. Objects are instances of classes. Classes define the structure and behaviour of their instances. Unlike many object-oriented systems, MMT classes are themselves objects whose structure and behaviour are defined by *meta*-classes. By providing access to meta-classes, MMT provides an open reflective architecture. For example, new objects are created by sending a 'new' message to a class whose meta-class defines how the message is handled.

MMT classes have a certain minimum structure and behaviour that is required for the basic features of object-orientation to work properly. These features include methods, attributes, object creation and inheritance. All features are implemented as objects that are instances of default MMT classes. Some of these features can be overriden in meta-classes. New features can be added by defining new meta-classes. MMT provides a basic class definition syntax that is engineered to accommodate the basic features.

This chapter describes the basic features of MMT classes.

Class Definitions

A class is created either by performing a class definition or by sending a meta-class a ‘new’ message. This section describes the basic features of class definitions.

The following package provides a examples of attributes, methods, single inheritance, run-super and object initialisation. The package defines a class representing two dimensional points. Two dimentional points have two attributes: the x and y position. The initial values of x and y are provided when a new point object is created. A point may be moved to a new position and its distance to the origin calculated:

```
1.  package SimpleClasses
2.      class Point
3.          x : Integer;
4.          y : Integer;
5.          init(s:Seq(Instance)):Object
6.              self.x := (s->at(0)) []
7.              self.y := (s->at(1)) []
8.              self
9.          end
10.         toString():String
11.             "(" + self.x + "," + self.y + ")"
12.         end
13.         move(x':Integer,y':Integer)
14.             self.x := x' []
15.             self.y := y'
16.         end
17.         dist():Integer
18.             ((self.x * self.x) + (self.y * self.y)).sqrt()
19.         end
20.     end
```

A circle is an extension of a point. The class Circle inherits from the class Point. Inheritance is declared using the **extends** clause of a class definition. If no inheritance is specified then a class inherits from Object. By inheriting from Point, the class Circle includes all of the attribute and method definitions from Point:

```
21.     class Circle extends SimpleClasses::Point
22.         radius : Integer;
```

```
23.         init(s:Seq(Instance)):Object
24.             super.run(s) []
25.             self.radius := (s->at(2)) []
26.             self
27.         end
28.         toString():String
29.             "<Circle " + super.run() + " r = " + self.radius + ">"
30.         end
31.     end
32. end
```

Lines 23 - 27 show an example of method redefinition. The class Circle inherits all of the method definitions from Point. Circle may choose to redefine any inherited methods. The body of a redefined method may refer to the inherited definition using the method call super.run(args). An example of run super is shown on line 24 where the initialisation method of Circle makes use of the initialisation method of Point. Line 29 shows how the Point toString method is called from the Circle toString method.

Instantiation

An instance of a class is created by sending the class a ‘new’ message. The MMT meta-class object creation protocol defines two methods for object creation:

1. Classifier::new():Object
   Sending a class a ‘new’ message with no arguments will create and return a new instance in which all the attributes are initialised to their default values.
2. Class::new(initArgs:Seq(Instance)):Object
   Sending a class a ‘new’ message with a sequence of initialisation arguments will create and return a new instance in which all the attributes are initialised to their default values and then instance is sent an ‘init’ message with the initialisation arguments. The default Object::init() method does nothing. Classes may redefine this to initialise each new instance on a class-by-class basis.

Invariants

A class has a number of invariants. Invariants are inherited. An invariant is a condition that must hold for all instances of the class in all stable states[1]. Invariants are useful for a number of development tasks:

1. Specifying a class. The invariant captures logical properties of the structure and behaviour of a class without the developer having to state how the structure and behaviour is realised.
2. Developing test cases for a class. The invariant expresses logical properties that must be true in all stable states. Tests can be generated from the invariant that can be run against objects.
3. Performing run-time checking. Invariants can be used during development to check the design of object structure and behaviour; they can be performed at various strategic points in the execution of a system to check object integrity.

The MMT class definition allows invariants to be expressed as boolean expressions after the keyword **inv**. Each invariant has a name, a boolean expression and a fail clause. A fail clause is a string expression that is evaluated in the context of an instance of the class when the invariant fails. The fail clause is used to provide diagnostic information.

Suppose that the age of a person is represented as an attribute and must be > 0:

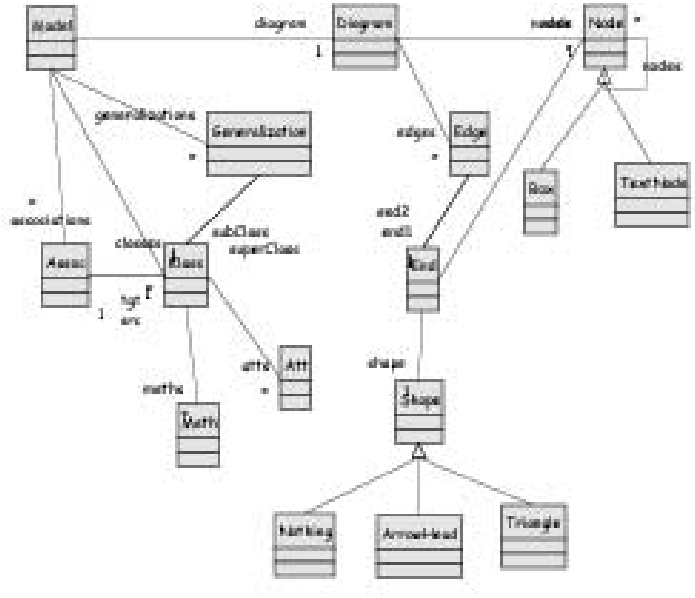
```
1. class Person
2.         age : Integer;
3.         inv
4.                 AgeGreaterThan0
5.                         age > 0
6.                         fail: self.age + " is not > 0"
7.                 end
8. end
```

Given an instance p of Person, the invariant can be checked in the following ways:

1. p.edit() creates an object editor window for p. An object editor includes a panel describing the outcome of running all the invariants for the object.

1. A stable state for an object is defined by the designer of the object's class. Often, an object should be in a stable state when it is not in the middle of handling a message.

2. p.check() performs all the invariants for p and returns a set of strings. If the set is empty then the invariants are all satisfied for the current state of p. Otherwise the set contains a set of diagnostic strings (created from the fail clauses of the invariants).
3. Person.check(p) checks the instance p against the invariants of the class Person. The result is a set of strings as described for 2 above.



The following example shows a more substantial use of invariants. Consider a simple modelling language consisting of classes, associations, attributes, methods and inheritance. Instances of such a language can be displayed on UML-like class diagrams. The rules determining whether a diagram correctly represents a model can be expressed as an invariant.

```
1.  package Invariants
2.      class Diagram
3.          nodes : Set(Invariants::Node);
4.          edges : Set(Invariants::Edge);
5.          init(s:Seq(Instance)):Object
6.              self.nodes := (s->at(0)) []
```

```
7.              self.edges := (s->at(1)) []
8.              self
9.          end
10.     end
```

Lines 1 -10 introduce the example package and define the class Diagram. A diagram is a graph structure: nodes are things like boxes and text and edges link nodes together. An edge has two ends attached to nodes, an edge has a label and has shapes (such as triangles and arrow heads) at either end.

```
11.     class Node
12.         x : Integer;
13.         y : Integer;
14.         height : Integer;
15.         width : Integer;
16.         nodes : Set(Invariants::Node);
17.          init(s:Seq(Instance)):Object
18.              self.x := (s->at(0)) []
19.              self.y := (s->at(1)) []
20.              self.height := (s->at(2)) []
21.              self.width := (s->at(3)) []
22.               self.nodes := (s->at(4)) []
23.               self
24.          end
25.         above(n:Invariants::Node):Boolean
26.              self.x = n.x and
27.              self.y + self.height = n.y
28.         end
29.     end
30.     class TextNode extends Invariants::Node
31.         text : String;
32.         init(s:Seq(Instance)):Object
33.              super.run(s) []
34.              self.text := (s->at(5)) []
35.              self
36.         end
37.         inv NoChildren self.nodes = Set{} fail: "No children allowed!"
38.     end
```

```
39.    class Box extends Invariants::Node end
```

Lines 11 - 24 define the class Node. A node is placed at an (x,y) position on a grid and has a height and width (the point (0,0) is at the top left of the grid). A node is also a node container allowing nodes to be nested. The predicate named above checks whether two nodes are placed on top of each other on a diagram.

Lines 30 - 37 define a sub-class of Node called TextNode. A text node displays some text and has no children. Line 38 defines a sub-class of Node called Box. A box draws a rectangle around its children.

```
40.    class Edge
41.        label : String;
42.        end1 : Invariants::End;
43.        end2 : Invariants::End;
44.        init(s:Seq(Instance)):Object
45.            self.label := (s->at(0)) []
46.            self.end1 := (s->at(1)) []
47.            self.end2 := (s->at(2)) []
48.        self
49.        end
50.    end
51.    class End
52.        node : Invariants::Node;
53.        shape : Invariants::Shape;
54.        init(s:Seq(Instance)):Object
55.            self.node := (s->at(0)) []
56.            self.shape := (s->at(1)) []
57.            self
58.        end
59.    end
60.    class Shape end
61.    class Nothing extends Invariants::Shape end
62.    class Triangle extends Invariants::Shape end
63.    class ArrowHead extends Invariants::Shape end
```

Lines 40 - 50 define the class Edge. An edge consists of two ends and a label (if the label is not displayed then it is the empty string). Lines 51 - 59 define the class EdgeEnd. The end of an edge is attached to a node and has a shape. Lines 60 - 63

define the shape classes. Shape is an abstract class, Nothing is used when no shape is to be displayed.

```
64.   class Model
65.       classes : Set(Invariants::Klass);
66.       associations : Set(Invariants::Assoc);
67.       generalizations : Set(Invariants::Generalization);
```

Lines 64 - 67 defines the class that represents models in the modelling language. A model consists of classes[2], associations between the classes and generalizations between the classes (inheritance). To simplify the example, associations are directed from a source class to a target class.

```
68.       diagram : Invariants::Diagram;
```

The diagram associated with a model is represented as an attribute of the model on line 6. For the purposes of this example we assume that a model is created interactively using a diagram tool. As the diagram is created, the model is populated. The two need to be consistent. The model will be used by other tools (for example translating to program code).

```
69.       addClass(c:Invariants::Klass)
70.           self.classes := (self.classes->including(c))
71.       end
72.       getClass(n:String):Invariants::Klass
73.           self.classes->select(c | c.name = n).selectElement()
74.       end
75.       addGen(g:Invariants::Generalization)
76.           self.generalizations := (self.generalizations->including(g))
77.       end
78.       addAssoc(a:Invariants::Assoc)
79.           self.associations := (self.associations->including(a))
80.       end
81.       setDiagram(d:Invariants::Diagram)
82.           self.diagram := d
```

2. We use the name Klass rather than Class so that the name does not clash with the MMT class with the same name. Although packages are name spaces, a number of classes are available everywhere; it is possible to shadow these classes but the mechanisms are somewhat fiddly. In general it is best to avoid names such as Class, Attribute, Package and Method.

```
83.         end
```

Lines 69 - 83 define a number of methods that set up a model.

```
84.         inv
85.             CheckDiagram
86.                 self.classes->forAll(c |
87.                     self.diagram.nodes->exists(n | c.isDiagram(n))) and
88.                 self.generalizations->forAll(g |
89.                     self.diagram.edges->exists(e | g.isDiagram(e))) and
90.                 self.associations->forAll(a |
91.                     self.diagram.edges->exists(e | a.isDiagram(e)))
92.                 fail: "Diagram is incorrect"
93.             end
94.     end
```

Lines 84 - 94 complete the class Model by defining the invariant named CheckDiagram. In order for a diagram to be consistent with a model all the classes must be represented as class boxes (lines 86 - 87), all generalizations must be represented as arrows from the sub-class to the super-class (lines 88 - 89) and all associations must be represented as arrows from the source class to the target class (lines 90 - 91).

```
95.     class Klass
96.         name : String;
97.         atts : Set(Invariants::Att);
98.         meths : Set(Invariants::Meth);
99.         init(s:Seq(Instance)):Object
100.            self.name := (s->at(0)) []
101.            self.atts := (s->at(1)) []
102.            self.meths := (s->at(2)) []
103.            self
104.        end
```

Lines 95 - 104 introduce a class that represents classes in a model. A class is checked against a diagram node using the method named isDiagram:

```
105.    isDiagram(node:Invariants::Node):Boolean
106.            node.nodes->size = 3 and
107.            node.nodes->exists(n1 |
108.                node.nodes->exists(n2 |
```

```
109.                    node.nodes->exists(n3 |
110.                        n1 <> n2 and
111.                        n2 <> n3 and
112.                        n1.above(n2) and
113.                        n2.above(n3) and
114.                        self.nameBox(n1) and
115.                        self.attsBox(n2) and
116.                        self.methsBox(n3))))
117.        end
```

A class is represented by a diagram node (line 105). The node should have three sub-nodes that it contains (line 106). There should be three distinct sub-nodes (n1, n2 and n3 on lines 107 - 111) such that the nodes are arranged one above the other (lines 112 and 113) where the top box contains the name of the class, the middle box contains the attributes of the class and the bottom box contains the methods of the box.

```
118.        nameBox(n:Invariants::Node):Boolean
119.            if n.isKindOf(Invariants::Box)
120.            then if n.nodes->size = 1
121.                then  if n.nodes.selectElement().isKindOf(Invariants::TextNode)
122.                      then n.nodes.selectElement().text = self.name
123.                      else false
124.                      endif
125.                else false
126.                endif
127.            else false
128.            endif
129.        end
```

Lines 118 - 129 define how a class checks that its name is displayed correctly. A box on a diagram is a rectangle drawn around its contents. A text node displays some text. A name box is a rectangle around the name of the class.

```
130.         attsBox(n:Invariants::Node):Boolean
131.            n.isKindOf(Invariants::Box) and
132.            self.atts->forAll(a |
133.                n.nodes->exists(n |
134.                    a.isDiagram(n)))
135.         end
```

```
136.          methsBox(n:Invariants::Node):Boolean
137.              n.isKindOf(Invariants::Box) and
138.              self.meths->forAll(m |
139.                  n.nodes->exists(n |
140.                      m.isDiagram(n)))
141.          end
142.      end
```

Lines 130 - 142 define checks for class attributes and methods. Both are displayed as boxes containing the type signatures of the appropriate features.

```
143.      class Generalization
144.          subClass : Invariants::Klass;
145.          superClass : Invariants::Klass;
146.          init(s:Seq(Instance)):Object
147.              self.subClass := (s->at(0)) []
148.              self.superClass := (s->at(1)) []
149.              self
150.          end
151.          isDiagram(edge:Invariants::Node):Boolean
152.              self.subClass.isDiagram(edge.end1.node) and
153.              self.superClass.isDiagram(edge.end2.node) and
154.              edge.end2.shape.isKindOf(Invariants::Triangle)
155.          end
156.      end
```

Lines 143 - 156 define how generalizations are represented in models. The predicate isDiagram on lines 151 - 155 requires the corresponding diagram edge to link the diagrammatic representations of the sub-class and super-class and for the end attached to the super-class to have a triangle shape.

```
157.      class Assoc
158.          name : String;
159.          src : Invariants::Klass;
160.          tgt : Invariants::Klass;
161.          init(s:Seq(Instance)):Object
162.              self.name := (s->at(0)) []
163.              self.src := (s->at(1)) []
164.              self.tgt := (s->at(2)) []
165.              self
```

```
166.        end
167.        isDiagram(edge:Invariants::Node):Boolean
168.            self.src.isDiagram(edge.end1.node) and
169.            self.tgt.isDiagram(edge.end2.node) and
170.            edge.end2.shape.isKindOf(Invariants::ArrowHead) and
171.            edge.label = self.name
172.        end
173.    end
```

The class Assoc on lines 157 - 173 holds between a source class and a target class. The association is named. The predicate named isDiagram is satisfied by an edge that links the corresponding diagrammtic representations of the associated classes and has an arrow head at the target class end.

```
174.    class Att
175.        name : String;
176.        type : String;
177.        init(s:Seq(Instance)):Object
178.            self.name := (s->at(0)) []
179.            self.type := (s->at(1)) []
180.            self
181.        end
182.        isDiagram(node:Invariants::Node):Boolean
183.            if node.isKindOf(Invariants::TextNode)
184.            then node.text = self.name + ":" + self.type
185.            else false
186.            endif
187.        end
188.    end
189.    class Meth
190.        name : String;
191.        args : String;
192.        type : String;
193.        init(s:Seq(Instance)):Object
194.            self.name := (s->at(0)) []
195.            self.args := (s->at(1)) []
196.            self.type := (s->at(2)) []
197.            self
198.        end
```

```
199.          isDiagram(node:Invariants::Node):Boolean
200.              if node.isKindOf(Invariants::TextNode)
201.              then node.text = self.name + self.args + ":" + self.type
202.              else false
203.              endif
204.          end
205.      end
```

Lines 174 - 205 define attributes and methods in models. Their diagram predicates are similar: both require the associated node to be a text node containing a text representation of the model element's type signature.

The following simple test suite completes the invariants package. It shows a model and its associated diagram. The model consists of two classes named c1 and c2. c2 inherits from c1. c1 defines an attribute named a1 and a method named m1. c2 defines an attribute named a2. An association named links from c1 to c2:

```
206.      test()
207.          let  a1 = Invariants::Att.new(Seq{"a1","Integer"})
208.               a2 = Invariants::Att.new(Seq{"a2","Boolean"})
209.               m1 = Invariants::Meth.new(Seq{"m1","(Integer)","Integer"})
210.               c1 = Invariants::Klass.new(Seq{"c1",Set{a1},Set{m1}})
211.          c2 = Invariants::Klass.new(Seq{"c2",Set{a2},Set{}})
212.          l = Invariants::Assoc.new(Seq{"a",c1,c2})
213.          m = Invariants::Model.new(Seq{})
214.          t1 = Invariants::TextNode.new(Seq{10,10,10,50,Set{},"c1"})
215.          t2 = Invariants::TextNode.new(Seq{10,20,10,50,Set{},"a1:Integer"})
216.          t3 = Invariants::TextNode.new(Seq{10,30,10,50,Set{},"m1(Integer):Integer"})
217.          t4 = Invariants::TextNode.new(Seq{10,100,10,50,Set{},"c2"})
218.          t5 = Invariants::TextNode.new(Seq{10,110,10,50,Set{},"a2:Boolean"})
219.          t6 = Invariants::TextNode.new(Seq{10,120,10,50,Set{},""})
220.          b1 = Invariants::Box.new(Seq{10,10,10,50,Set{t1}})
221.          b2 = Invariants::Box.new(Seq{10,20,10,50,Set{t2}})
222.          b3 = Invariants::Box.new(Seq{10,30,10,50,Set{t3}})
223.          n1 = Invariants::Node.new(Seq{10,10,30,50,Set{b1,b2,b3}})
224.          b4 = Invariants::Box.new(Seq{10,100,10,50,Set{t4}})
225.          b5 = Invariants::Box.new(Seq{10,110,10,50,Set{t5}})
226.          b6 = Invariants::Box.new(Seq{10,120,10,50,Set{t6}})
227.          n2 = Invariants::Node.new(Seq{10,100,20,50,Set{b4,b5,b6}})
```

```
228.        e1 = Invariants::End.new(Seq{n2,Invariants::Nothing.new(Seq{})})
229.        e2 = Invariants::End.new(Seq{n1,Invariants::Triangle.new(Seq{})})
230.        e3 = Invariants::End.new(Seq{n1,Invariants::Nothing.new(Seq{})})
231.        e4 = Invariants::End.new(Seq{n2,Invariants::ArrowHead.new(Seq{})})
232.        e = Invariants::Edge.new(Seq{"",e1,e2})
233.        e' = Invariants::Edge.new(Seq{"a",e3,e4})
234.        d = Invariants::Diagram.new(Seq{Set{n1,n2},Set{e,e'}})
235.    in  m.addClass(c1) []
236.        m.addClass(c2) []
237.        m.addAssoc(l) []
238.        m.addGen(Invariants::Generalization.new(Seq{
239.            m.getClass("c2"),m.getClass("c1")})) []
240.        m.setDiagram(d) []
241.        m
242.    end
243. end
244.end
```

The Structure and Behaviour of Classes

MMT classes are objects and as such have structure and behaviour. The class interface is split into two parts: the *classifier* interface and the *class* interface. A classifier is an object that has instances which are not necessarily objects. Examples of classifiers are Boolean and Integer. Instances of classifiers have behaviour and invariants but no slots. A class is an object that is a classifier that defines attributes; each attribute becomes a slot in the instances of the class, therefore instances of classes are objects.

MMT defines two meta-classes: Classifier and Class. Class defines the basic features necessary to be a classifier. Class inherits from Classifier and adds the extra features necessary to be a class. This section defines the classifier and class interfaces and provides examples of their use.

The Classifier Interface

This section defines the interface of the MMT class Classifier.

```
1.  class Classifier extends NameSpace
```

```
2.    initialised : Boolean ;
3.    owner : Classifier ;
4.    allMethods : Seq(BehaviouralFeature) ;
5.    methods : Seq(BehaviouralFeature) ;
6.    allConstraints : Set(Constraint) ;
7.    invariant : Set(Constraint) ;
8.    isAbstract : Boolean ;
9.    allParents : Set(Classifier) ;
10.   parents : Set(Classifier) ;
11.   generator : Closure ;
12.   docString : String ;
13.   name : String ;
14.   id : String ;
```

Lines 1 - 14 introduce the class Classifier and define its attributes. All classifiers must be initialised (line 2) at most once; this is performed by calling the init method. Usually you will not call the init method directly; it is performed automatically when a class or package definition of performed or indirectly when a classifier is created by sending a new message to a meta-class.

All classifiers have an owner (line 3) that is a classifier. If a classifier is contained in a package then the package is the owner. If the classifier is not contained then it is its own owner.

Classifiers define methods (lines 4 and 5). The locally defined methods of a classifier are its methods, all the methods defined and inherited by the classifier are allMethods. The order in which methods occur in these sequences is important since method lookup will invoke the first one found. Usually you should use the appropriate methods to add and remove methods to and from a classifier.

Classifiers define invariant constraints (lines 6 and 7). The locally defined constraints of a classifier are its invariant and all the constraints defined and inherited are allConstraints. Usually you should use te appropriate methods to add and remove constraints to and from and a classifier.

A classifier is abstract when it has no instances (line 8). This is currently not used.

Classifiers inherit methods and constraints from their parents (lines 9 and 10). The locally defined super-classifiers are parents and the transitive closure of the parents relation is allParents.

The generator of a classifier (line 11) is part of the MMT implementation and should not be used.

A classifier has a documentation string (line 12) that describes its purpose and use. A classifier has a name (line 13). A classifier has a unique identifier (line 14).

15. init(s : Seq(Instance)):Object

A classifier can be created by sending the class Classifier or one of its sub-classes a new message. The initialisation arguments (line 15) should be a sequence of elements as follows: Seq{name,doc,parents,isAbstract}. All other components of a classifier should be added usingthe appropriate classifier methods.

16. new():Instance

The class Classifier defines the instantiation protocol for MMT instances (line 16). All classifiers can be instantiated with no initialisation arguments. The result is a new instance of the classifier. By default this method is abstract: it must be given suitable definitions by sub-classes of Classifier.

17. allLocalParents():Set(Classifier)

18. allInheritedParents():Set(Classifier)

19. allParents():Set(Classifier)

A classifier c has parents from which it inherits structure and behaviour. The transitive closure of the parents relationship is computed by c.allLocalParents() (line 17). In addition, c indirectly receives parents from its owner. Any classifiers defined by c.owner (either locally or via inheritance) named c.name are computed by c.allInheritedParents(). Finally, the complete set of parents for c is the union of the local and inherited parents c.allParents() (line 19). Usually, you should use the method allParents to acces the parents of a classifier.

20. inheritsFrom(c : Classifier):Boolean

The method inheritsFrom (line 20) can be used to test whether the receiver inherits from another classifier.

21. default():Instance

All classifiers must specify a default value (line 21) that is used when a slot is initialised whose corresponding attribute type is the classifier. The default default value is the class Instance.

22. allConstraints():Set(Constraint)

All the constraints defined and inherited by a classifier are computed by the method allConstraints (line 22). You should use this method to access all the constraints rather than the slot allConstraints.

23. checkInstance(instance : Instance):Set(String)

A classifier defines a collection of constraints that define how it classifies its instances. The method checkInstance (line 23) is used to determine whether a given candidate instance is correctly classified. The method runs all of the constraints against the candidate and returns the set of diagnostic strings. Note that checking os performed in super-class to sub-class order. If checking fails in a given class then it does not progress to further classes (since failure in a super-class may mean that it is not possible to perform subsequent tests).

24. allMethods():Seq(Method)

All the methods defined and inherited by a classifier are computed by the method allMethods (line 24). You should use this in preference to the slot allMethods.

25. getMethod(name : String, arity : Integer):Method
26. removeMethod(name : String, arity : Integer):Method
27. addMethod(method : Method):Method
28. hasMethod(name : String, arity : Integer):Boolean

Lines 25 - 28 define methods that can be used to access methods defined by a classifier.

29. hasConstraint(name : String):Boolean
30. removeConstraint(name : String):Constraint
31. getConstraint(name : String, default : Instance):Instance
32. addInvariant(constraint : Constraint):Constraint

Lines 29 - 32 define methods that can be used to access constraints defined by a classifier.

33. toDescription():String
34. toFlatDescription():String
35. **end**

Lines 33 - 34 define two methods that are used to produce a textual definition of a classifier. The method toDescription produces the original definition of the classifier. The method toFlatDescription produces a flattened definition by including all of the inherited methods anc constraints.

Methods

This section defines the interface of the MMT class Method. Method is part of a family of classes rooted at Behaviour::BehaviouralFeature. The class Behavioural-Feature defines the essential components of an object that can be invoked by message passing. The interface is as follows:

```
1.  class BehaviouralFeature
2.      knownToVM : Boolean ;
3.      owner : Classifier ;
4.      type : Classifier ;
5.      types : Seq(Classifier) ;
6.      args : Seq(String) ;
7.      name : String ;
8.      init(s : Seq(Instance)):Object
9.      send(target : Instance,args : Seq(Instance)):Instance
10.     apply(args : Seq(Instance)):Instance
11. end
```

A behavioural feature consists of a name (line 7), a sequence of argument names (line 6), a sequence of argument types (line 5), a return type (line 4) and an owner (line 5).

The boolean attribute named knownToVM (line 2) determines whether or not the feature is directly executable by the MMT virtual machine. Methods defined in package or class definitions are always known to the VM. You may define your own sub-classes of BehaviouralFeature in order to define new message passing protocols. In this case you must define a method named send (line 9) that the VM will invoke on the target and arguments of the message. In addition, a method may be used as a function and applied to arguments (line 10). The default message passing protocol is to report an error and the deault behavioural application protocol is to send a message to the feature.

The initialisation arguments for a behavioural feature are:

Seq{name,args,types,type,owner}

The class Method implements a concrete behavioural feature.

```
12. class Method extends Behaviour::BehaviouralFeature
13.     body : Closure ;
```

```
14.     docString : String ;
15.     init(s : Seq(Instance)):Object
16.     send(target : Instance, args : Seq(Instance)):Object
17. end
```

You may create a method directly and subsequently add it to a classifier. The initialisation arguments for the method (line 10) are:

Seq{name,args,types,type,owner,doc,body}

The body of a method is a function whose arity matches the number of arguments defined for the method. The messag epassing protocol of a method (line 16) invokes the body on the message arguments. If the body of a method instance is updated then this will have effect the next time the method is invoked.

Constraints

This section defines the interface of the MMT class Constraint.

```
1. class Constraint
2.      body : Closure ;
3.      owner : Classifier ;
4.      name : String ;
5.      init(s:Seq(Instance)):Object
6. end
```

A constraint has a name (line 4), an owner (line 3) and a body (line 2). The body of a constraint is a function of one argument. To test a constraint its body is supplied with the candidate instance. The initialisation arguments for a constraint are:

Seq{name,owner,body}

The Class Interface

This section defines the interface of the MMT class Class.

```
1. class Class extends Classifier
2.      allAttributes : Set(Attribute) ;
3.      attributes : Set(Attribute) ;
4.      allAttributes():Set(Attribute)
5.      hasAttribute(name : String):Boolean
```

```
6.      findAttribute(name : String):Instance
7.      removeAttribute(name : String):Attribute
8.      addAttribute(a : Attribute):Attribute
9.  end
```

The class Class is a sub-class of Classifier (line 1). The local attributes are defined by the the attribute named attributes (line 3). The attributes locally defined and inherited are the value of allAttributes (line 2). You should use the method allAttributes (line 4) to calculate all the attributes.

Attributes

This section defines the interface of MMT attributes.

```
1.  class Attribute
2.      docString : String ;
3.      type : Classifier ;
4.      name : String ;
5.      init(s : Seq(Instance)):Object
6.  end
```

State Transition Machines

The following example shows how classes can be treated as objects. The example adds new attributes and methods to a class. In addition, the example implements a new type of behavioural feature together with its own message passing protocol.

A state transition machine consists of transitions between states and describes behaviour associated with a class of objects. A transition is labelled and contains both a guard condition and an action. A transition is enabled when an instance receives an event corresponding to the transition label, the instance is in the source state and the guard is satisfied by the current instance state. Given a collection of enabled transitions for an instance, the intended meaning is that one transition is selected at random and the associated action is performed on the instance; the instance makes a transition to the target state.

```
1.  open Structure;
2.  package Machines
3.      class Machine
```

```
4.      states : Set(String);
5.      trans : Set(Machines::Transition);
6.      init(s:Seq(Instance)):Object
7.          self.states := (s->at(0)) []
8.          self.trans := (s->at(1)) []
9.          self
10.     end
```

Line 1 is an **open** statement. It is used at the top of package files to place the names defined by a package in scope for the rest of the file. There may be any number of **open** statements at the top of a file. The the case of line 1 the package Structure is opened because we know that we will be using initialised attributes later in the package. Unlike Attribute the class InitialisedAttribute is not in scope by default (they are both defined in the package Structure).

```
11.     apply(args:Seq(Instance)):Instance
12.         let c = args->at(0)
13.         in   self.addStateAtts(c) []
14.              self.addTransMethods(c)
15.         end
16.     end
```

Supose that for convenience we wish to add the behaviour of a state machine M to a class C by *applying* the machine to the class: M(C). Any MMT value may be used as a function provided that it defines a method named apply. Lines 11 - 16 define the apply method for a machine. The arguments in the application are supplied to the method as a sequence (line 11); therefore, each argument (in this case c) must be extracted from the sequence (line 12).

To add the behaviour a state machine to a class we make the following assumptions. The states will be added as boolean attributes in the class. The transitions will be added as methods: the label on the transition gives the name of the method. Each transition method has exactly 1 argument (this simplifies the example since otherwise we would have a number of similar cases handling different numbers of arguments 2, 3, 4 etc.) The attributes are added on line 13 and the methods are added on line 14.

```
17.     addStateAtts(c:Class)
18.         self.states->collect(s |
19.             if c.hasAttribute(s)
20.             then state.error("Machine::addStateAtts: " + s + " already present.")
```

```
21.                 else c.addAttribute(Attribute.new(Seq{s,Boolean}))
22.                 endif)
23.         end
24.         addTransMethods(c:Class)
25.             let labels = self.trans->collect(t | t.label)
26.             in labels->collect(l |
27.                 c.addMethod(Machines::MachineMethod.new(Seq{l,c,self})))
28.             end
29.         end
```

Lines 17 - 29 define methods to add state attributes and transition methods to a class. Line 25 constructs the set of transition labels and then line 27 adds a method for each label. The method is an instance of the class MachineMethod defined below.

```
30.         send(target:Instance,message:String,args:Seq(Instance)):Instance
31.             let enabled = self.trans->select(t |
32.                     target.slotValue(t.src) and
33.                     t.label = message and
34.                     (t.guard)(target))
35.             in  if not enabled->isEmpty
36.                 then let choice = enabled.selectElement()
37.                     in choice(target,args)
38.                     end
39.                 else message + " was ignored: no transition enabled."
40.                 endif
41.             end
42.         end
```

A state machine defines the behaviour of an instance when the instance receieves a named event. Such an event occurs in response to sending the a message to a target instance. Lines 30 - 42 defines a method for handling such an event. Lines 31 - 33 construct a set of enabled transitions. A transtion is enabled when the target of the message is in the source state, the message is the same as the transition label and the guard on the transition is satisfied by the target.

If there is at least one enabled transition then one is selected at random (lines 35 and 36) and the transition is performed by applying it to the target and arguments. If no transition is enabled then a suitable message is returned (line 39.

```
43.          inv
44.              TransitionsBetweenLegalStates
45.                  self.transitions->forAll(t |
46.                      self.states->includes(t.src) and self.states->includes(t.tgt))
47.                  fail: "transitions must be between states in " + self.states.toString()
48.              end
49.      end
```

The invarint defined on lines 44 - 48 requires that the transitions in a state machine be defined between states defined for the machine. If this invariant is violated then machine behaviour can go wrong since a transition could test a state variable that does not exist.

```
50.      class MachineMethod extends Behaviour::BehaviouralFeature
51.          machine : Machines::Machine;
52.          init(s:Seq(Instance)):Object
53.              let   name = s->at(0)
54.                  owner = s->at(1)
55.                  machine = s->at(2)
56.              in self.machine := machine []
57.                  super.run(Seq{name,Seq{"arg"},Seq{Instance},Instance,owner})
58.              end
59.          end
60.          send(target:Instance,args:Seq(Instance)):Instance
61.              self.machine.send(target,self.name,args)
62.          end
63.      end
```

Lines 50 - 63 define a machine method class. Instances of this class are added to a class for each transition label. When a machine method is invoked it sends a message to machine which will perform the appropriate transition. Since a machine method has no individual body, the class MachineMethod is a sub-class of BehaviouralFeature. The method named send on lines 60 - 62 defines the behaviour of a machine method.

```
64.      class Transition
65.          src : String;
66.          tgt : String;
67.          label : String;
68.          guard : Closure;
```

```
69.         action : Closure;
70.         init(s:Seq(Instance)):Object
71.             self.src := (s->at(0)) []
72.             self.tgt := (s->at(1)) []
73.             self.label := (s->at(2)) []
74.             self.guard := (s->at(3)) []
75.             self.action := (s->at(4)) []
76.             self
77.          end
78.         apply(args:Seq(Instance)):Instance
79.             let   target = args->at(0)
80.                   messageArgs = args->at(1)
81.             in target.setSlot(self.src,false) []
82.                target.setSlot(self.tgt,true) []
83.                self.action.self := target []
84.                 (self.action)(messageArgs)
85.             end
86.          end
87.      end
```

Lines 64 - 87 define the class Transition. A transition has a label, a source and target state, a guard and an action. The guard is a predicate that expects to be supplied with an object. The action is a function that expects to be supplied with a sequence of arguments.

Line 83 sets the value of the special variable ‘self’ in the action function of the transition. Line 84 performs the action by applying the function to the supplied arguments.

```
88.      class DrinksMachine
89.          coins : Integer;
90.          drinks : Integer = 1;
91.          price : Integer = 10;
92.      end
```

Lines 88 - 92 define a simple class for a drinks machine. The drinks machine has a slot for coins and a button for dispensing drinks once the the correct amount of money has been inserted. The attribute coins is the amount of money currently

inserted; the attribute drinks is the number of individual drinks in the machine; the attribute price is the price of each drink.

```
93.     machine():Machines::Machine
94.         let t1 = Machines::Transition.new(Seq{
95.                     "Idle","Vending",
96.                     "insertCoins",
97.                     fun(m)
98.                         m.drinks > 0
99.                     end,
100.                    fun(args)
101.                        self.coins := (self.coins + args->at(0))
102.                    end})
```

The method named machine (line 93) constructs and returns a machine defining the behaviout of a drinks machine. The machine consists of three transitions. The first transition (lines 94 - 102) represents the initial coins inserted in the machine which changes from the idle state to the vending state. The guard on the transition requires the machine to be non-empty. The action increases the amount of money.

```
103.        t2 = Machines::Transition.new(Seq{
104.                    "Vending","Vending",
105.                    "insertCoins",
106.                    fun(m)
107.                        m.drinks > 0 and m.coins < m.price
108.                    end,
109.                    fun(args)
110.                        self.coins := (self.coins + args->at(0))
111.                    end})
```

The second transition (lines 103 - 111) defines the behaviour of the machine when coins are inserted during the vending state. The coins are accepted until the amount of money in the machine exceeds the price of a drink.

```
112.        t3 = Machines::Transition.new(Seq{
113.                    "Vending","Idle",
114.                    "pressButton",
115.                    fun(m)
116.                        m.coins >= m.price
117.                    end,
```

```
118.                    fun(args)
119.                        self.coins := (self.coins - self.price) []
120.                        self.drinks := (self.drinks - 1)
121.                    end})
122.            m = Machines.Machine.new(Seq{Set{"Idle","Vending"},Set{t1,t2,t3}})
123.        in m
124.        end
125.    end
```

The final transition (lines 112 - 121) handles the case when sufficient money has been inserted and the operator presses a button. The money is consumed and the drink is dispensed. The machine makes a transition to the idle state waiting further money.

```
126.    printMachine(m:Machines::DrinksMachine)
127.        ("coins = " + m.coins).println() []
128.        ("drinks = " + m.drinks).println() []
129.        ("idle = " + m.Idle).println() []
130.        ("vending = " + m.Vending).println() []
131.        "".println()
132.    end
133.    test()
134.        (Machines::machine())(Machines::DrinksMachine) []
135.        let m = Machines::DrinksMachine.new(Seq{})
136.        in  m.Idle := true []
137.            Machines::printMachine(m) []
138.            m.insertCoins(4) []
139.            Machines::printMachine(m) []
140.            m.insertCoins(7) []
141.            Machines::printMachine(m) []
142.            m.pressButton(0) []
143.            Machines::printMachine(m)
144.        end
145.    end
146. end
```

Lines 133 - 145 define a small test suite for the drinks machine. The machine behaviour is applied to the drinks machine class on line 134. The application

changes the class by side-effect. A new machine is created on line 135, the initial state of the machine is set to idle on line 136. Drinks cost 10 pence, the customer inserts 4 pence followed by 7 pence and then presses the button. The output is as follows:

```
147.    coins = 0
148.    drinks = 1
149.    idle = true
150.    vending = false
151.
152.    coins = 4
153.    drinks = 1
154.    idle = false
155.    vending = true
156.
157.    coins = 11
158.    drinks = 1
159.   idle = false
160.   vending = true
161.
162.   coins = 1
163.   drinks = 0
164.   idle = true
165.   vending = false
```

Inheritance and Method Combination

One way of viewing inheritance is as a reuse mechanism. Definitions in a super-class are reused when defining a sub-class. In particular methods in a super-class are available in the sub-class. Another way of viewing inheritance is as a *merging* mechanism for partial views of a type. Definitions given by a number of super-class are merged to produce the sub-class. This section describes how method merging is defined in MMT and how you can control method merging by defining new types of methods.

Consider single inheritance between a super-class C and a sub-class D. C defines a method named m. If D does not define a method named m then inheritance is straightforward: m is available in both C and D. If D defines a method named m

then there is a conflict when a message m is sent to an instance of D. MMT resolves this conflict by combining the two definitions to produce a single method which is then invoked.

By default the method combination rule in MMT are to invoke the most specific definition for a method in a left-to-right traversal of the inheritance lattice; and, to make methods occurring subsequently in this ordering available in the body of the invoked method by sending the special message 'run' to the object named 'super'.

The default combination rule allows flexibility since the most specific method (in the case of C and D, D::m) can choose whether to invoke the shadowed definition (C::m) and how to merge the two results. For example:

```
1.  class C
2.      m(args)
3.          // some body
4.      end
5.  end
6.  class D extends C
7.      m(args)
8.          if // some condition
9.          then super.run(exps)
10.         else // some expression
11.         endif
12.     end
13. end
```

The default rule allows a shadowing method to choose not to invoke the shadowed method, to invoke it once or to invoke it many times. If there are many classes linked together in an inheritance chain, each providing a definition for a method with the same name then the *run super* mechanism allows the individual methods to be chained together when they are invoked.

Although the default rule is very flexible, it is an imperitive mechanism and is therefore not especially *declarative*. To determine how methods are combined it is necessary to trace the internal execution path of method invocations; it is not possible to see what effect method combination will have and how the results of methods will be combined just from looing at the method definitions from the outside. In particular, since it is so imperative, it is often difficult to reason about the default combination rules in conjunction with multiple inheritance, especially if a class

multiply inherits from classes with a multiple common super-classes (the diamond import problem).

To support a more declarative form of method combination MMT has a number of builtin combination rules that can be used to override the default. Furthermore, new combination rules can be defined.

RunAll

When multiple inheritance is used to combine classes together, it is desirable for methods to be merged so that they are all performed when an appropriate message is sent to an instance. MMT provides a method combination rule called RunAll that allows methods to be declared as being components of a larger method; when they are merged they combine to form a single composite method.

Suppose that multiple inheritance is to be used to combine classes representing a married person and a teacher to implement a married teacher. The following package defines four classes: Person, Teacher and Married that both independently inherit from Person and MarriedPerson that multiply inherits from Person and Teacher.

Each class defines a toString method that is used to display its instances. Each toString method is a partial definition that is extended in sub-classes. When the toString method is multiply inherited by MarriedTeacher, the result is a composite method that combines the partial definitions from Married and Teacher.

```
1.  package RunAllExample
2.      class Person
3.          name : String;
4.          init(s:Seq(Instance)):Object
5.              self.name := (s->at(0)) [] self
6.          end
7.          toString():String
8.              "<" + self.of.name + self.toStringBody().separateWith(" ") + ">"
9.          end
10.         toStringBody():String metaclass(Behaviour::RunAll)
11.             " name = " + self.name
12.         end
13.     end
```

The class Person defines a toString method on lines 7 - 9 that uses an auxiliary method called toStringBody to transform the person slots to a string. The reason for the autiliary method is to allow for arbitrary extensions to the display in sub-classes.

The method toStringBody on lines 10 - 12 declares its metaclass to be the MMT class Behaviour::RunAll. By default the metaclass of a method is Behaviour::Method that uses the default method combination rule. If an alternative metaclass is specified then the method combination rule is given by its definition of 'send'.

The class Behaviour::SendAll combines methods by performing all the methods and returning their results as a sequence. The order in which the methods will be invoked is a depth first left to right traversal of the inheritance lattice; although in most cases this order should not be relevant and it is good design not to rely on it.

The method toString Body is defined to be of type Behaviour::RunAll because it is intended to be extended in sub-classes. Each extension is a partial definition of the method and all should be performed when an instance receives a toStringBody message.

The classes Teacher and Married both extend the class Person:

```
14.  class Teacher extends RunAllExample::Person
15.      subject : String;
16.      init(s:Seq(Instance)):Object
17.          self.name := (s->at(0)) []
18.          self.subject := (s->at(1)) []
19.          self
20.      end
21.      toStringBody():String metaclass(Behaviour::RunAll)
22.          " subject = " + self.subject
23.      end
24.  end
25.  class Married extends RunAllExample::Person
26.      spouse : String;
27.      init(s:Seq(Instance)):Object
28.          self.name := (s->at(0)) []
29.          self.spouse := (s->at(1)) []
30.          self
```

```
31.      end
32.      toStringBody():String metaclass(Behaviour::RunAll)
33.          " spouse = " + self.spouse
34.      end
35.  end
```

Both Teacher and Married[3] provide partial definitions of toStringBody on lines 21 - 23 and 32 - 34. Note that the metaclass in both cases is RunAll: all the metaclasses for the same method name should be consistent in classes related by inheritance.

```
36.      class MarriedTeacher
37.          extends
38.              RunAllExample::Married,
39.              RunAllExample::Teacher
40.          init(s:Seq(Instance)):Object
41.              let  name = s->at(0)
42.                   spouse = s->at(1)
43.                   subject = s->at(2)
44.              in   RunAllExample::Married::init.send(self,Seq{Seq{name,spouse}}) []
45.                   RunAllExample::Teacher::init.send(self,Seq{Seq{name,subject}}) []
46.                   self
47.              end
48.          end
49.      end
50.  end
```

Lines 36 - 49 define a class MarriedTeacher that inherits from both Married and Teacher. The initialisation method directly calls the initialisation methods of the two super-classes on lines 44 and 45. The display method for MarriedTeacher is constructed automatically by the method combination rules so, for example, the following object:

```
RunAllExample::MarriedTeacher.new(Seq{"fred","wilma","rockBreaking"})
```

3. The init methods of both Teacher and Married repeat the initialisation performed by Person. A better design decision would havebeen to use super.run(s) in both cases rather than self.name := (s->at(0)). Unfortunately a bug in MMT then prevents the initialisation methods being invoked directly as in Married::init.send(self,Seq{name,spouse}) (see later). So for the purposes of this example super.run is not used.

is displayed as:

```
<MarriedTeacher subject = rockBreaking  spouse = wilma  name = fred>
```

CHAPTER 6

Packages

MMT packages are containers of definitions. A package may contain class, association, method and package definitions. A package provides a name space for its contents. The main purpose of packages is to provide a grouping mechanism for system components. Typically a package contains all the classes and associations relating to a system module. A package may contain methods that are used to create, test and animate instances of the package contents. Where a system module naturally decomposes into a number of sub-modules its package can be structured in terms of a number of nested sub-packages.

Packages can be specialised using inheritance. Package specialisation allows a sub-package to inherit the contentsof the parent package. This mechanism supports incremental system development, modularity and reuse. Systems can be developed incrementally by constructing a minimal collection of packages, testing them and then extending them with extra functionality using package specialisation. Systems are modular since they can be expressed as the composition of a number of modules. Each module has a well defined interface and may be replaced with a different module with minimal changes to client modules. System development can take advantage of reuse by extending from a library of modules that are shared over a wide range of applications.

MMT supports a novel form of package specialisation where definitions in packages are *virtual* in the sense of C++. A package P may contain a definition D that is

referenced throughout P. A package Q specialises P and specialises the definition D (perhaps D is a class and Q adds some structure or behaviour to D). Q will inherit all of the definitions in P including all references to D; however, references to D in Q will result in the specialised D.

The MMT package specialisation mechanism provides a very flexible mechanism for modular system development. Many different aspects of a system can be developed as independent packages. Each package can be tested independently. The complete system is defined as the combination of the parts represented as a single package that specialises all the part-packages. Package specialisation will ensure that intra-package references are resolved correctly.

Packages are MMT objects. They may be manipulated just like any other object in the system. This provides a very flexible development environment in which developers can implement their own system tools. For example a development team may have a particular strategy for applying regression tests on all the modules in an application. Such a strategy can easily be developed in MMT by constructing a method that iterates over all the application packages, applying a suite of tests to the classes and recording the results.

The structure and behaviour of packages are defined by the MMT class Package. Since MMT is an open environment, the class Package is available to the developer and may be specialised to support application specific structure and behaviour.

This chapter defines package definitions, the package interface and provides some examples of package use.

The Package Definition

Packages can be created in the following two ways:

1. By performing a package definition. A package definition occurs in MML text and is of the form **package** ... **end**.
2. By sending a ‘new’ message to the class Package.

The following is a simple package definition:

```
1. package P
2.     class C
```

```
3.          x : P::D;
4.      end
5.      class D
6.          y : P::C;
7.      end
8.  end
```

Lines 1 - 9 define a package named P. The package contains two classes named C and D. The class C has an attribute named x of type D and the class D has an attribute named y of type C. The package P provides a name space. The name P is scoped over the name space of P allowing the definitions for C and D to refer to each other.

Given the definition of P above, the class C is referred to as P::C and similarly P::D. Therefore, a new instance of C is created as P::C.new(Seq{}).

Animals

Packages may be used without specialisation to structure systems. If a package name P is currently in scope then any of its contained definitions D are available as P::D. In addition to structuring, packages can be used for incremental system definition using package specialisation. This section gives a simple example of a collection of animals. The types of animals are built up incrementally using package specialisation: a basic package of animal types is defined; the basic package is specialised in two different ways by adding new types of animals and extended existing animal types; finally, the two specialisations are merged by defining a package that specialises both.

```
1.  open Associations;
2.  package Animals
3.      class Animal
4.          name : String;
5.          toString():String
6.              "<" + self.of.name + " " + self.name + ">"
7.          end
8.      end
9.      class Person extends Animals::Animal  end
10.     class Dog extends Animals::Animal
```

```
11.          inv
12.              NotMr Pink
13.                  self.owner.name <> "Mr. Pink"
14.                  fail: "Mr Pink is not allowed to own dogs."
15.              end
16.      end
17.      association Ownership
18.          owns: Animals::Animal mult: *;
19.          owner: Animals::Person mult: 1;
20.      end
21.      class Cat extends Animals::Animal end
22.  end
```

Lines 1 - 22 define a package named Animals. Cats and dogs are owned by people although Mr. Pink has been banned from owning a dog.

```
23. open Associations;
24. open Structure;
25. package NoisyAnimals extends Animals
26.      class Parrot extends NoisyAnimals::Animal
27.          stockPhrase : String = “Polly wants a cracker!”;
28.      end
29.      class Cat
30.          purrVolume : Integer = 3;
31.      end
32.      class Dog
33.          barkVolume : Integer = 5;
34.      end
35.      association FightsWith
36.          enemies: NoisyAnimals::Cat mult: *;
37.          enemies: NoisyAnimals::Dog mult: *;
38.      end
39. end
```

Lines 23 - 39 define a specialisation of the package Animals. A new class named Parrot is introduced. The Parrot class extends the Animal class on line 26; note that since NoisyAnimals extends Animals, the class Animal is referenced with respect to the name space NoisyAnimals (line 26).

The classes Cat and Dog are extended in NoisyAnimals. Since they are inherited from Animals there is no need to use an **extends** clause on lines 29 and 32. A new association between Cat and Dog is added on lines 35 - 38. The association references the extended definitions via the NoisyAnimals name space.

```
40. open Structure;
41. package GreedyAnimals extends Animals
42.     class Animal
43.         food : String;
44.     end
45.     class Donkey extends GreedyAnimals::Animal end
46.     class Cow extends GreedyAnimals::Animal end
47.     class Dog
48.         favouriteFood : String = “Doggy Chow”;
49.     end
50. end
```

The class Animal is extended in GreedyAnimals to record the food that an animal eats (line 43). New types of animal are added: Donkey and Cow. The class Dog is extended with favourite food; the most popular form of dog food is specified as a default value.

```
51. package CompleteAnimals
52.  extends GreedyAnimals, NoisyAnimals
53.     class Animal
54.         age : Integer;
55.         inv
56.             AgeGreaterThan0
57.                 self.age >= 0
58.                 fail: "The age of animals must be >= 0."
59.             end
60.     end
61. end
```

Lines 51 - 61 define a package that extends both greedy and noisy animals. The definition of classes Animal, Dog and Cat in in both GreedyAnimals and NoisyAnimals are merged into single definitions in CompleteAnimals. The default merging rules currently select the left-most definition for attributes and methods defined in the same class.

The Package Interface

An MMT package is an instance of the MMT class Package whose interface is defined below:

```
1.  class Package extends Classifier
2.      contents : Set(Classifier) ;
3.      allContents():Set(Classifier)
4.      allMergedContents():Set(Classifier)
5.      classes():Set(Class)
6.      packages():Set(Package)
7.      associations():Set(BinaryAssociation)
8.      allClasses():Set(Class)
9.      allPackages():Set(Package)
10.     allAssociations():Set(BinaryAssociation)
11.     add(c : Classifier):Classifier
12.     remove(c : Classifier):Classifier
13. end
```

A package is a classifier and therefore a name space. Since it is a classifier (line 1) it has instances each of which is a snapshot. A package contains definitions: local definitions are those directly defined by the package and inherited definitions arise indirectly through its parents. The local definitions are the contents of a package (line 2). The method allContents (line 3) returns the complete set of contents of a package: both local and inherited.

Since a package may have multiple parents, its allContents may contain two or more definitions with the same name. The method allMergedContents (line 4) returns a set of definitions where no name occurs more than once: multiple definitions with the same name are *merged*. The merging rules for packages and merge their contents. Where packages do not have multiple parents or do not have multiple contents with the same name then the result of allContents and allMergedContents are the same.

The methods classes, packages and associations (lines 5 - 7) return the local definitions of the appropriate category. The methods allClasses, allPackages and allAssociations (lines 8 - 10) return all the merged definitions of the appropriate category.

The methods add and remove are used to change the local contents of a package.

A Robot Command Language

Package specialisation can be used to support a separation of concerns during development. It is often possible in an application to be decomposed into control modules and services modules. Package provide a mechanism by which these aspects of an application can be developed independently and then merged using-package specialisation.

This section provides a simple example of an applcation that is developed as three packages: a module tht implements a control language; a module that implements a collection of services; and a module that implements the control language in terms of the services.

A robot lives on a grid of cells. Each cell may contain a stack of cones. The job of the robot is to move around the grid moving cones between cells. At any instant in time the robot is facing in one of the dorections: north; south; east; west. The robot may turn 90 degrees left or right, may move forwards (if possible), may pick up a cone from the current cell (is available) and may drop a cone onto the current cell (if carried).

The world of the robot can be implemented as a package of services called Grids:

```
1.  open Structure;
2.  open Stacks;
3.  package Robots
4.      package Grids
5.          class Grid
6.              width : Integer;
7.              height : Integer;
8.              cells : Set(Grids::Cell);
9.              init(s:Seq(Instance)):Object
10.                 self.width := (s->at(0)) []
11.                 self.height := (s->at(1)) []
12.                 self.initCells() []
13.                 self
14.             end
```

The class named Grid implements the world inhabited by a robot. A grid is initialised with a width and a height (lines 6,7 and 10, 11), the set of cells (line 8) is initialised to contain a cell at each co-ordinate (line 12):

```
15.     initCells()
16.         (0).to(self.height)->collect(y |
17.             (0).to(self.width)->collect(x |
18.                 self.addCell(x,y)))
19.     end
20.     addCell(x:Integer,y:Integer)
21.         self.cells := (self.cells->including(Grids::Cell.new(Seq{x,y})))
22.     end
```

The cell found at a given co-ordinate position is found using the method cellAt:

```
23.     cellAt(x:Integer,y:Integer):Grids::Cell
24.         self.cells->select(cell | cell.x = x and cell.y = y).selectElement()
25.     end
```

A grid is displayed by constructing a string that has a character representing each cell; the grid string is constructed by iterating throught the cells adding a newline character at the end of each row:

```
26.     toString():String
27.         (0).to(self.height)->iterate(y output = "" |
28.             (0).to(self.width)->iterate(x output = output |
29.                 output + self.cellAt(x,y).contentString()) + "\n")
30.     end
```

The position of an item in the grid is found by iterating through the cells:

```
31.     position(item:Grids::Item):Grids::Cell
32.         if self.inGrid(item)
33.         then self.cells->select(cell | cell.contains(item)).selectElement()
34.         else state.error("Grid::position: item not in grid " + item.toString())
35.         endif
36.     end
37.     inGrid(item:Grids::Item):Boolean
38.         self.cells->exists(cell | cell.contains(item))
39.     end
```

When an item is dropped onto a grid it is added to the cell at the given location:

```
40.     dropAt(item:Grids::Item,x:Integer,y:Integer)
41.         self.cellAt(x,y).drop(item)
```

```
42.          end
43.      end
```

Cells exist at a particular position in a grid and have contents. The contents are organised as a stack: each time a new value is dropped onto a cell it is pushed onto the stack of contents. Only the top element on a cell is available.

```
44.      class Cell
45.          x : Integer;
46.          y : Integer;
47.          contents : Stack = Stack.new();
48.          init(s:Seq(Instance)):Object
49.              self.x := (s->at(0)) []
50.              self.y := (s->at(1)) []
51.              self
52.          end
53.          contentString():String
54.              cond
55.                  self.contents.empty() then ".";
56.                  else self.contents.top().toString()
57.              end
58.          end
59.          empty():Boolean
60.              self.contents.empty()
61.          end
62.          top():Grids::Item
63.              self.contents.top()
64.          end
65.          drop(item:Grids::Item)
66.              self.contents.push(item)
67.          end
68.          pop()
69.              self.contents.pop()
70.          end
71.          contains(item:Grids::Item):Boolean
72.              self.contents.contains(item)
73.          end
74.      end
```

Cells contain items. An item is either a cone or a robot. If a cell contains a robot then, by convention, the robot should be the top cell element:

```
75.     class Item
76.         toString():String
77.             "!"
78.         end
79.     end
80.     class Cone extends Grids::Item
81.         toString():String
82.             "A"
83.         end
84.     end
```

A robot is facing in a given direction and is carrying a collection of items:

```
85.     class Robot extends Grids::Item
86.         direction : String = "north";
87.         items : Set(Grids::Item);
88.         toString():String
89.             cond
90.                 self.direction = "north" then "^";
91.                 self.direction = "south" then "V";
92.                 self.direction = "west" then "<";
93.                 self.direction = "east" then ">";
94.                 else state.error("Robot::toString: unknown direction " +
95.                         self.direction)
96.             end
97.         end
```

A robot can grab an item:

```
98.          grab(item:Grids::Item)
99.             self.items := (self.items->including(item))
100.         end
```

A robot can turn left or right:

```
101.        left()
102.            cond
103.                self.direction = "north" then self.direction := "west";
```

```
104.                    self.direction = "south" then self.direction := "east";
105.                    self.direction = "west" then self.direction := "south";
106.                    self.direction = "east" then self.direction := "north";
107.                    else state.error("Robot::left: unknown direction " + self.direction)
108.                end
109.            end
110.            right()
111.                cond
112.                    self.direction = "north" then self.direction := "east";
113.                    self.direction = "south" then self.direction := "west";
114.                    self.direction = "west" then self.direction := "north";
115.                    self.direction = "east" then self.direction := "south";
116.                    else state.error("Robot::right: unknown direction " + self.direction)
117.                end
118.            end
```

A robot may move forward one cell:

```
119.            forward(grid:Grids::Grid)
120.                let cell = grid.position(self)
121.                in  cond
122.                        self.direction = "north" then
123.                            if cell.y > 0
124.                            then cell.pop() [] grid.dropAt(self,cell.x,cell.y-1)
125.                            endif;
126.                        self.direction = "south" then
127.                            if cell.y < grid.height
128.                            then cell.pop() [] grid.dropAt(self,cell.x,cell.y+1)
129.                            endif;
130.                         self.direction = "west" then
131.                            if cell.x > 0
132.                            then cell.pop() [] grid.dropAt(self,cell.x-1,cell.y)
133.                            endif;
134.                        self.direction = "east" then
135.                            if cell.x < grid.width
136.                            then cell.pop() [] grid.dropAt(self,cell.x+1,cell.y)
137.                            endif
138.                    end
139.                end
```

```
140.        end
```

The forward method uses position (line 12) to find the cell containing the robot. The conditional expression (lines 121 - 138) is used to analyse the current direction that the robot is facing. In each case the the cell position is checked to ensure that forward movement is legal. If the robot can make the move then it is removed from the cell (cell.pop()) and dropped at the new location.

A robot can pick up the item at the head of the pile at its current location:

```
141.        pickup(grid:Grids::Grid)
142.            let cell = grid.position(self)
143.            in  cell.pop() []
144.                if cell.empty()
145.                then cell.drop(self)
146.                else self.grab(cell.top()) []
147.                     cell.pop() []
148.                     cell.drop(self)
149.                endif
150.            end
151.        end
```

By convention the robot is always the top element on the pile of items at its current location. It is removed from the location (line 143) to reveal the pile underneath. If the cell is then non-empty the top item is grabbed and removed from the cell and the robot is added back.

A robot may drop one of the items it is carrying. For the purposes of this example, if the robot is carrying more than one item then one is selected at random:

```
152.        drop(grid:Grids::Grid)
153.             if not self.items->isEmpty
154.            then let cell = grid.position(self)
155.                      item = self.items.selectElement()
156.                 in  cell.pop() [] cell.drop(item) [] cell.drop(self) []
157.                     self.items := (self.items->excluding(item))
158.                 end
159.            endif
160.        end
161.    end
```

```
end
```

The package Grids provides a collection of services that may be used by suitable controller packages. Consider the case where a controller is to be designed that uses a language of control commands to manipulate a robot in terms of the services offered by Grids. The control command language can be separated into its *syntax* and its *semantics*. The syntax defines appearance and structure of the control commands; for the purposes of this example we present the abstract syntax. The semantics defines how each command is performed using the services offered by Grids.

The abstract syntax of the control command language is defined by the package Commands:

```
package Commands
    class Command end
    class Nothing extends Commands::Command end
    class Left extends Commands::Command end
    class Right extends Commands::Command end
    class Forward extends Commands::Command end
    class Drop extends Commands::Command end
    class Pickup extends Commands::Command end
    class Then extends Commands::Command
        left : Commands::Command;
        right : Commands::Command;
        init(s:Seq(Instance)):Object
            self.left := (s->at(0)) []
            self.right := (s->at(1)) []
            self
        end
    end
    program(commands:Seq(Commands::Command)):Commands::Command
        commands->iterate(command c = Commands::Nothing.new(Seq{}) |
            Commands::Then.new(Seq{c,command}))
    end
    repeat(command:Commands::Command,n:Integer):Commands::Command
        (1).to(n)->iterate(_ c = Commands::Nothing.new(Seq{}) |
            Commands::Then.new(Seq{c,command}))
    end
end
```

All commands are peformed with respect to a robot and a grid. The command Nothing does nothing; Left and Right turn the robot 90 degrees in the appropriate direction; Forward moves the robot one cell forward if possible; Pickup grabs the top item from the cell occupied by the robot if available; Drop leaves one of the items carried by the robot on top of its cell if available.

The package Commands provides an example of a language that is tailored specifically to an application domain. We are using a toy domain, however in general the domain may be some aspect of a large application for example the business rules or the control or the physical deployment. Each different aspect of an application at each level of refinement is best expressed using a language that is tailored for the purpose.

The package Commands defines the abstract syntax of a command language. Other features that could be defined in Commands are the display of commands or a command editor. The packages Commands and Grids are composed using package specialisation; for each command we must define the services that are used to implement it. This is done in the following package named Controller:

```
189.  package Controller
190.      extends
191.          Robots::Commands,
192.          Robots::Grids
193.      class Command
194.          perform(robot:Controller::Robot,grid:Controller::Grid)
195.              state.error("Command::perform is abstract.")
196.          end
197.      end
```

The package Controller extends the definitions from both Commands and Grids (lines 190 - 192). The class Command is extended with a method named ‘perform’ (lines 193 - 197) that adds semantics to each command: it must be re-defined on a case-by-case basis.

```
198.      class Nothing
199.          perform(robot:Controller::Robot,grid:Controller::Grid)
200.              grid
201.          end
202.      end
```

A Nothing command is performed (lines 199 - 201) by leaving the robot in its current state with respect to the grid.

```
203.    class Left
204.        perform(robot:Controller::Robot,grid:Controller::Grid)
205.            robot.left()
206.        end
207.    end
208.    class Right
209.        perform(robot:Controller::Robot,grid:Controller::Grid)
210.            robot.right()
211.        end
212.    end
213.    class Forward
214.        perform(robot:Controller::Robot,grid:Controller::Grid)
215.            robot.forward(grid)
216.        end
217.    end
218.    class Drop
219.        perform(robot:Controller::Robot,grid:Controller::Grid)
220.            robot.drop(grid)
221.        end
222.    end
223.    class Pickup
224.        perform(robot:Controller::Robot,grid:Controller::Grid)
225.            robot.pickup(grid)
226.        end
227.    end
228.    class Then
229.        perform(robot:Controller::Robot,grid:Controller::Grid)
230.            self.left.perform(robot,grid) [] self.right.perform(robot,grid)
231.        end
232.    end
233. end
```

Lines 203 - 233 show the semantics of each command. In most cases they simply call the appropriate method of Robot, supplying the current state of the grid. The exception is Then (lines 228 - 232) that performs the left command before the right command.

The package Robots is completed with a simple test method that creates a grid, drops two cones and a robot and then performs a simple program that moves the robot around the grid and moves a cone from one cell to another:

```
234.    test()
235.        let grid = Robots::Controller::Grid.new(Seq{5,7})
236.            cone1 = Robots::Controller::Cone.new(Seq{})
237.            cone2 = Robots::Controller::Cone.new(Seq{})
238.            robot = Robots::Controller::Robot.new(Seq{})
239.            left = Robots::Controller::Left.new(Seq{})
240.            forward = Robots::Controller::Forward.new(Seq{})
241.            right = Robots::Controller::Right.new(Seq{})
242.            pickup = Robots::Controller::Pickup.new(Seq{})
243.            drop = Robots::Controller::Drop.new(Seq{})
244.            repeat = Robots::Controller::repeat
245.            program = Robots::Controller::program(Seq{
246.                left,
247.                repeat(forward,4),
248.                right,
249.                forward,
250.                right,
251.                repeat(forward,3),
252.                pickup,
253.                forward,
254.                drop,
255.                left,
256.                forward})
257.        in  grid.toString().println() []
258.            grid.dropAt(cone1,0,0) []
259.            grid.dropAt(cone2,4,6) []
260.            grid.dropAt(robot,5,7) []
261.            grid.toString().println() []
262.            program.perform(robot,grid) []
263.            grid.toString().println()
264.        end
265.    end
266. end
```

The command Robots::test() produces the following output:

```
......
......
......
......
......
......
......
......


A.....
......
......
......
......
......
....A.
.....^


A.....
......
......
......
......
.....^
.....A
......
```

Filmstrips

The Robot Command Language defined in the previous section provides an example of separating a package of services from a language that uses those services. The commands were given an *imperative* semantics using package specialisation by defining a package called Controller that defines methods for each type of command.

Suppose that we have a language that expresses dynamic behaviour; UML has such languages in the form of statecharts and collaboration diagrams. The language in question is to be used to design part of the dynamic behaviour of a system and as such we would like to test candidate executions of parts of the system against the

design. For such a scenario, an imperative semantics like the one we developed for the robot command language is not appropriate because we have no way of representing executions and therefore no way of constructing one and presenting it to MMT to test.

Executions of systems are sometimes referred to as *filmstrips.* The idea of a filmstrip is that is captures a step-by-step development of system execution in terms of states; each step in the filmstrip describes a change in state. Although there are many different data representations that may be used for filmstrips, including trees and graphs, a simple representation is a sequence that encodes a *linear filmstrip*.

For example, a linear filmstrip for the robot system would be a sequence of grids linked with a record of the operations that caused one grid to change into the next.

This section uses packages to develop a simple filmstrip application for integer arithmetic. We develop a package of arithmetic expressions and an imperative state machine that can be used to perform the expressions. A filmstrip for the language is a sequence of machine states.

The package containing the example is called Eval:

```
1.  open Stacks;
2.  open Structure;
3.  package Eval
4.      package Exps
5.          class Exp end
6.          class Number extends Exps::Exp
7.              value : Integer;
8.              init(s:Seq(Instance)):Object
9.                  self.value := (s->at(0)) []
10.                 self
11.             end
12.             toString():String
13.                 self.value.toString()
14.             end
15.         end
16.         class Add extends Exps::Exp
17.             left : Exps::Exp;
18.             right : Exps::Exp;
```

```
19.             init(s:Seq(Instance)):Object
20.                 self.left := (s->at(0)) []
21.                 self.right := (s->at(1)) []
22.                 self
23.             end
24.             toString():String
25.                 self.left.toString() + "+" + self.right.toString()
26.             end
27.         end
28. end
```

Lines 4 - 28 define a package of arithmetic expressions called Exps. For the purposes of brevity the expressions are limited to numeric constants (lines 6 - 15) and addition expressions (lines 16 - 27).

The package Machine defines the states that will occur in filmstrips and that will be used in the imperative operational semantics of the expression language:

```
29.     package Machine
30.         class State
31.             stack : Stack = Stack.new(Seq{});
32.             control : Stack = Stack.new(Seq{});
33.             init(s:Seq(Instance)):Object
34.                 if s->isEmpty
35.                 then self
36.                 else self.stack := (s->at(0)) []
37.                     self.control := (s->at(1)) []
38.                     self
39.                 endif
40.             end
41.             toString():String
42.                 "(" + self.stack.toString() + "," + self.control.toString() + ")"
43.             end
```

A machine state (lines 31 and 32) consists of a value stack and a control stack. Expressions and machine instructions live on the control stack. The results of performing expressions (i.e. integers) live on the value stack. When the machine performs an expression, the expressions and instructions are consumed from the top of

the control stack. If the resulting control item required operands then they are found at the head of the value stack.

```
44.         loadExp(e:Eval::Exps::Exp):Machine::State
45.             self.control.push(e) []
46.             self
47.         end
48.         loadVal(n:Integer):Machine::State
49.             self.stack.push(n) []
50.             self
51.         end
```

Expressions and values are loaded onto the appropriate stacks using the methods loadExp and loadVal defines on lines (44 - 51).

```
52.             add(other:Machine::State):Machine::State
53.                 Machine.State.new(Seq{
54.                     self.stack + other.stack,
55.                     self.control + other.control})
56.             end
57.         end
```

Two machine states can be added together to produce a new machine state (lines 52 - 56). In adding together machine states their corresponding stacks are added together.

```
58.         class AddInstr
59.             toString():String
60.                 "+"
61.             end
62.         end
63.     end
```

In addition to performing expressions, the machine has its own instruction set. Since the expression language is small, the instruction set is also small. A single instruction is defined (lines 58 - 63) that will add up the top two integers on the value stack.

The operational meaning of the language can be defined using an imperative semantics. This allows us to animate expressions in our language. Not all languages that employ the idea of filmstrips lend themselves to defining such an oper-

ational semantics because their meaning is highly ambiguous (meaning that each expression leads to a very large number of possible filmstrips). This does not apply here so we can define a machine as follows:

```
64.  package Operational extends Eval::Exps, Eval::Machine
65.      class State
66.          perform()
67.              if self.control.empty()
68.              then self.stack.pop()
69.              else self.control.pop().perform(self) [] self.perform()
70.              endif
71.          end
72.      end
```

The package Operational extends both Exps and Machine (line 64) in order to add methods named 'perform' to state and expression classes. The definition of perform encodes the semantics of the language.

The perform method for states (lines 66 - 72) continually pops a control item and performs it with respect to the current machine state. When the control is exhausted, the machine terminates returning the top value stack item.

```
73.      class Number
74.          perform(s:Operational::State)
75.              s.stack.push(self.value)
76.          end
77.      end
```

When a number is performed (lines 73 - 77) the value of the number is pushed onto the head of the value stack.

```
78.      class Add
79.          perform(s:Operational::State)
80.              s.control.push(Operational::AddInstr.new(Seq{})) []
81.              s.control.push(self.left) []
82.              s.control.push(self.right)
83.          end
84.      end
```

When an addition is performed, a new add instruction is pushed on the control (line 80) followed by the left and right sub-expressions (lines 81 and 82). A design

choice arises here regarding the order in which the sub-expressions should be performed. The machine cannot perform both sub-expressions concurrently since it is a sequential machine. However, the expression language is free from side effects so performing the sub-expressions in either order should be acceptable. We will return to this issue when we look at filmstrips for the expression language.

```
85.         class AddInstr
86.             perform(s:Operational::State)
87.                 s.stack.push(s.stack.pop() + s.stack.pop())
88.             end
89.         end
```

An add instruction is performed by adding the two items at the top of the value stack (line 87). The top two items are replaced by their sum.

```
90.         eval(e:Operational::Exp):Integer
91.             Operational::State.new(Seq{}).loadExp(e).perform()
92.         end
93.     end
```

The Operational package concludes with a useful method named ‘eval’ (lines 90 - 92). This method is used to perform an expression by loading it onto a fresh machine state, running the machine and then returning the result of the expression.

The package Operational defines how to perform an expression. Suppose instead we want to test a candidate expression evaluation to see if it conforms to the semantics of expressions. Since the language we have developed is small, we can do this by generating the set of filmstrips for an expression and then testing the candidate to see if it belongs to the set[1].

```
94.     package Filmstrips extends Eval::Exps
95.         class Filmstrip
96.             states : Seq(Eval::Machine::State);
97.                 init(s:Seq(Instance)):Object
98.                 self.states := s []
99.                 self
100.            end
```

1. Of course this technique will only work if the set of possible filmstrips for an expression is small. On the other hand it is a viable way of defining all the legal filmstrips even if other techniques are used to test candidate expression executions.

```
101.        toString():String
102.            "<Filmstrip " + self.states->iterate(s S = Seq{} |
103.                    S + Seq{s.toString()}).separateWith(",") + ">"
104.        end
```

Lines 94 - 104 introduce the package Filmstrips. A filmstrip is a sequence of machine states.

Two filmstrips can be added together. This is best explained using an example. Suppose f1 is the filmstrip (using a shorthand notation):

[([],[n1]),([n1],[])]

and f2 is the filmstrip:

[([],[n2]),([n2],[])]

then f1 + f2 is the filmstrip:

[([],[n1+n2]),([],[n1,n2,+]),([n1],[n2,+]),([n2,n1],[+]),([n3],[])]

where n3 = n1+n2. The filmstrip method add is defined as follows:

```
105.        add(right:Filmstrips::Filmstrip):Filmstrips::Filmstrip
106.            let lastLeft = self.states->last
107.                firstLeft = self.states.head
108.                leftExp = firstLeft.control.top()
109.                leftVal = lastLeft.stack.top()
```

The resulting filmstrip describes the result of performing the receiver and then the calculation described by the filmstrip right (line 105), Lines 106 - 109 extract some required components from the left hand filmstrip. The expression leftExp (line 108) is the first sub-expression to be performed producing the value leftVal (line 109).

```
110.                lastRight = right.states->last
111.                firstRight = right.states.head
112.                rightExp = firstRight.control.top()
113.                rightVal = lastRight.stack.top()
```

Lines 110 - 113 extract the corresponding components from the right filmstrip that is to be performed after the left filmstrip.

```
114.        preState = Eval::Machine::State.new(Seq{})
115.        postState = Eval::Machine::State.new(Seq{})
```

Lines 114 and 115 define two new states that will be the starting state and final state of the new filmstrip. The starting state contains an empty value stack and has the the addition of the left and right expressions as the single entry on the control stack. The final state has the addition of the left and right values as the single entry on the value stack and has an empty control stack.

```
116.        addInstr = Eval::Machine::AddInstr.new(Seq{})
117.        delayedRight =
118.            let s = Eval::Machine::State.new(Seq{})
119.            in s.loadExp(rightExp)
120.            end
121.        delayedInstr =
122.            let s = Eval::Machine::State.new(Seq{})
123.            in s.loadExp(addInstr)
124.            end
125.        performedLeft =
126.            let s = Eval::Machine::State.new(Seq{})
127.            in s.loadVal(leftVal)
128.            end
```

Lines 116 - 128 define three new states. The state delayedRight will be added to the states in the left filmstrip to record the pending right sub-expression. The state perfomedLeft will be added to the states in the right filmstrip to record the value produced by the left filmstrip. The state delayedInstr will be added to the states in both left and right filmstrips to record the pending + instruction.

```
129.        leftMergedStates = self.states->collect(s |
130.            s + delayedRight + delayedInstr)->asSequence
131.        rightMergedStates = right.states->collect(s |
132.            s + performedLeft + delayedInstr)->asSequence
133.        mergedStates = leftMergedStates + rightMergedStates.tail
```

Lines 129 - 133 merges the states form the left and right filmstrips. The states in the left filmstrip are modified by adding the delayed right sub-expression and the pending + instruction. The states in the right filmstrip are modified by adding the result from the left filmstrip and the pending + instruction. The merged states are the result of appending the two filmstrips. We drop the head of the right merged

states because the last element of the right merged filmstrips and the head of the right merged filmstrips are the same state.

```
134.                in  preState.loadExp(Eval::Exps::Add.new(Seq{leftExp,rightExp})) []
135.                    postState.loadVal(leftVal + rightVal) []
136.                    Filmstrips::Filmstrip.new(
137.                        Seq{preState} + mergedStates + Seq{postState})
138.                end
139.            end
140.        end
```

Lines 134 - 139 construct the final merged filmstrip. The pre state and the post state are modified by adding in the + expression and its result. Finally, lines 136 - 137 create and return a new filmstrip.

Each expression produces a set of filmstrips describing all possible ways of performing the expression. In particular an addition expression can be performed two ways: performing the left then performing the right or performing the right then performing the left.

```
141.        class Number
142.            calculations():Set(Filmstrips::Filmstrip)
143.                let pre = Eval::Machine::State.new(Seq{})
144.                    post = Eval::Machine::State.new(Seq{})
145.                in  pre.loadExp(self) []
146.                    post.loadVal(self.value) []
147.                    Set{Filmstrips::Filmstrip.new(Seq{pre,post})}
148.                end
149.            end
150.        end
```

Lines 141 - 150 defines the calculations produced by a Number. A number calculation has the following shape:

```
[([],[n]),([n],[])]
```

```
151.            class Add
152.                calculations():Set(Filmstrips::Filmstrip)
153.                    self.addFilmstrips(self.left,self.right)->union(
154.                        self.addFilmstrips(self.right,self.left))
```

```
155.                end
```

The filmstrips produced by an addition expression is the union of the calculations produced by performing left then right and performing right then left (lines 153 and 154).

```
156.                addFilmstrips(left:Filmstrips::Exp,right:Filmstrips::Exp):
157.                    Set(Filmstrips::Filmstrip)
158.                let leftFilmstrips = left.calculations()
159.                    rightFilmstrips = right.calculations()
160.                in leftFilmstrips->iterate(l C = Set{} |
161.                        rightFilmstrips->iterate(r C = C |
162.                            C->including(l + r)))
163.                end
164.            end
165.        end
166.    end
```

Lines 156 - 166 define a method that adds left and right sub-expressions of an addition expression. The result is a set of filmstrips containing the pair-wise concatenation of left and right filmstrips (lines 160 - 162).

Given an expression e, the set e.calculations() contains all possible ways of performing the expression on the state transition machine. Given a candidate filmstrip we could test whether the candidate is correct by checking whether or not it is contained in the set.

```
167.  test()
168.      let n = Eval::Operational::Number.new(Seq{10})
169.          add = Eval::Operational::Add.new(Seq{n,n})
170.          n' = Eval::Filmstrips::Number.new(Seq{10})
171.          add' = Eval::Filmstrips::Add.new(Seq{n',n'})
172.          add" = Eval::Filmstrips::Add.new(Seq{n',add'})
173.      in  Eval::Operational::eval(add).toString().println() []
```

Lines 167 - 173 define a test for the Eval package. Line 173 prints the integer 20 on the standard output.

```
174.          n'.calculations().toString().println() []
```

Line 174 prints the following to the standard output:

```
Set{<Filmstrip (<Stack >,<Stack 10>),(<Stack 10>,<Stack >)>}
```

```
175.        add'.calculations().toString().println() []
```

Line 175 prints the following to the standard output:

```
Set{<Filmstrip (<Stack >,<Stack 10+10>),
                (<Stack >,<Stack 10,10,+>),
                (<Stack 10>,<Stack 10,+>),
                (<Stack 10,10>,<Stack +>),
                (<Stack 20>,<Stack >)>,
    <Filmstrip (<Stack >,<Stack 10+10>),
                (<Stack >,<Stack 10,10,+>),
                (<Stack 10>,<Stack 10,+>),
                (<Stack 10,10>,<Stack +>),
                (<Stack 20>,<Stack >)>}
```

```
176.        add".calculations().toString().println()
177.      end
178.    end
179. end
```

Line 176 prints the following to the standard output:

```
Set{<Filmstrip (<Stack >,<Stack 10+10+10>),
                (<Stack >,<Stack 10,10+10,+>),
                (<Stack 10>,<Stack 10+10,+>),
                (<Stack 10>,<Stack 10,10,+,+>),
                (<Stack 10,10>,<Stack 10,+,+>),
                (<Stack 10,10,10>,<Stack +,+>),
                (<Stack 20,10>,<Stack +>),
                (<Stack 30>,<Stack >)>,
    <Filmstrip (<Stack >,<Stack 10+10+10>),
                (<Stack >,<Stack 10,10+10,+>),
                (<Stack 10>,<Stack 10+10,+>),
                (<Stack 10>,<Stack 10,10,+,+>),
                (<Stack 10,10>,<Stack 10,+,+>),
                (<Stack 10,10,10>,<Stack +,+>),
                (<Stack 20,10>,<Stack +>),
                (<Stack 30>,<Stack >)>,
    <Filmstrip (<Stack >,<Stack 10+10+10>),
                (<Stack >,<Stack 10+10,10,+>),
                (<Stack >,<Stack 10,10,+,10,+>),
```

```
                (<Stack 10>,<Stack 10,+,10,+>),
                (<Stack 10,10>,<Stack +,10,+>)
                (<Stack 20>,<Stack 10,+>),
                (<Stack 10,20>,<Stack +>),
                (<Stack 30>,<Stack >)>,
<Filmstrip (<Stack >,<Stack 10+10+10>),
                (<Stack >,<Stack 10+10,10,+>),
                (<Stack >,<Stack 10,10,+,10,+>),
                (<Stack 10>,<Stack 10,+,10,+>),
                (<Stack 10,10>,<Stack +,10,+>),
                (<Stack 20>,<Stack 10,+>),
                (<Stack 10,20>,<Stack +>),
                (<Stack 30>,<Stack >)>}
```

Calculations

The linear filmstrips used in the previous section describe how a state transition machine performs a given expression. This raised problems regarding the order in which sub-expressions were performed. Sometimes it may be more appropriate to be abstract with respect to the ordering of sub-expression evaluation. This section shows how expression evaluation can be expressed as *calculations* and how candidate calculations can be checked against expressions.

A calculation is a graph. The nodes of the graph represent data values (in this case integers) and the edges of the graph represent operations that are performed on data values (in this case addition). An edge has many source nodes; each source node is a required input for the operation. An edge has many target nodes; each target node is data produced by the operation (in this example we only require one target node).

Calculations are general structures that can be used to encode a wide range of language execution. In this example the language is the package Exps defined in the previous section:

```
1. open Stacks;
2. open Structure;
3. package Calculations
4.     package Exps
5.         // as defined in the section named Filmstrips
6.     end
```

Calculations are defined by the package Calcs:

```
7.      package Calcs
8.          class Node
9.              data : Integer;
10.             init(s:Seq(Instance)):Object
11.                 self.data := (s->at(0)) []
12.                 self
13.             end
14.             toString():String
15.                 "<Node " + self.data + ">"
16.             end
17.             equals(other:Calcs::Node):Boolean
18.                 other.data = self.data
19.             end
20.         end
```

A calculation contains nodes that represent data (lines 8 - 20).

```
21.         class Edge
22.             label : String;
23.             sources : Set(Calcs::Node);
24.             targets : Set(Calcs::Node);
25.             init(s:Seq(Instance)):Object
26.                 self.label := (s->at(0)) []
27.                 self
28.             end
29.             toString():String
30.               "(" + self.sources->collect(s |
31.                 s.toString())->asSequence.separateWith(",") + ") -" + self.label + "->(" +
32.                     self.targets->collect(s |
33.                         s.toString())->asSequence.separateWith(",") + ")"
34.             end
35.             addSource(n:Calcs::Node)
36.                 self.sources := (self.sources->including(n))
37.             end
38.             addTarget(n:Calcs::Node)
39.                 self.targets := (self.targets->including(n))
40.             end
```

A calculation contains edges that represent operations. Each edge has a set of source nodes (line 23) and a set of target nodes (line 24). The methods addSource and addTarget (lines 35 - 40) allow source and target nodes to be added to an edge.

```
41.         equals(other:Calcs::Edge):Boolean
42.             self.label = other.label and
43.             self.subSources(other) and other.subSources(self) and
44.             self.subTargets(other) and other.subTargets(self)
45.         end
46.         subSources(other:Calcs::Edge):Boolean
47.             self.sources->forAll(n1 | other.sources->exists(n2 | n1.equals(n2)))
48.         end
49.         subTargets(other:Calcs::Edge):Boolean
50.             self.targets->forAll(n1 | other.targets->exists(n2 | n1.equals(n2)))
51.         end
52.     end
```

Lines 41 - 52 define that two edges are equal when they have the same source and target nodes.

```
53.     class Calc
54.         nodes : Set(Calcs::Node);
55.         edges : Set(Calcs::Edge);
56.         toString():String
57.             "<Calc " + self.nodes.toString() + "," + self.edges.toString() + ">"
58.         end
59.         init(s:Seq(Instance)):Object
60.             if s->size = 2
61.             then let nodes = s->at(0)
62.                      edges = s->at(1)
63.                  in self.nodes := nodes []
64.                     self.edges := edges []
65.                     self
66.                  end
67.             else self
68.             endif
69.         end
70.         addNode(n:Calcs::Node)
71.             self.nodes := (self.nodes->including(n))
```

```
72.         end
73.         addEdge(e:Calcs::Edge)
74.             self.edges := (self.edges->including(e))
75.         end
```

Lines 53 - 75 introduce the class Calc. A calculation consists of sets of nodes and edges. A calculation may be constructed by supplying its node and edge sets (lines 59 - 69); these will default to Set{} if none are supplied. New nodes and edges are added to a calculation using the methods addNode and addEdge (lines 70 - 75).

```
76.         add(other:Calcs::Calc):Calcs::Calc
77.             let c = Calcs::Calc.new(Seq{})
78.             in  c.nodes := (self.nodes->union(other.nodes)) []
79.                 c.edges := (self.edges->union(other.edges)) []
80.                 c
81.             end
82.         end
```

Two calculations are merged by adding them together (lines 76 - 82). The result is a new calculation consisting of the union of the edge and node sets from the component calculations.

```
83.         equals(other:Calcs::Calc):Boolean
84.             self.subCalc(other) and
85.             other.subCalc(self)
86.         end
87.         subCalc(other:Calcs::Calc):Boolean
88.             self.nodes->forAll(n1 | other.nodes->exists(n2 | n1.equals(n2))) and
89.             self.edges->forAll(e1 | other.edges->exists(e2 | e1.equals(e2)))
90.         end
```

Lines 83 - 90 define when one calculation is equal to another. Equality is defined in terms of a sub-calc relation (lines 87 - 90) that is satisfied when the nodes and edge sets are subsets.

```
91.         outputFringe():Set(Calcs::Node)
92.             self.nodes->select(node |
93.                 not self.edges->exists(edge | edge.sources->includes(node)))
94.         end
```

The output fringe of a calculation (lines 91 - 94) is the set of nodes representing the data values produced by the calculation. The output fringe is important when determining the value(s) produced by a calculation and when merging calculations together.

```
95.     resultNode():Calculations::Calcs::Node
96.         let outputs = self.outputFringe()
97.         in  if outputs->size = 1
98.             then outputs.selectElement()
99.             else state.error("Calc::resultNode: more than one output " +
100.                 outputs.toString())
101.            endif
102.        end
103.    end
104.    result():Integer
105.        self.resultNode().data
106.    end
```

Lines 95 - 106 define methods that assume a calculation has a single output (for the purposes of this example all calculations will have a single output). The result node (lines 95 - 103) is the node that is not the source of any edge. The result of a calculation (lines 104 - 106) is the data in the result node.

```
107.    subCalcs():Set(Calcs::Calc)
108.        let  nodeSets = self.nodes.power()
109.             edgeSets = self.edges.power()
110.        in nodeSets->iterate(nodes S = Set{} |
111.            edgeSets->iterate(edges S = S |
112.                let calc = Calcs::Calc.new(Seq{nodes,edges})
113.                in  if calc.wellFormed()
114.                    then S->including(calc)
115.                    else S
116.                    endif
117.                end))
118.        end
119.    end
```

A calculation has a set of well-formed sub-calculations (lines 107 - 119). A sub-calculation is formed by taking a sub-set of the nodes and edges. The sub-calculation is well formed when all the nodes are the source or target of some edge and all

the sources and targets of edges are also in the calculation. Sub-calculations are formed by combining the elements of the power sets of the nodes and edges (lines 108 -112). The resulting calculations are then filtered using a well formedness check (line 113).

```
120.            wellFormed():Boolean
121.                if self.outputFringe()->size = 1
122.                then
123.                    if self.nodes->forAll(node |
124.                        self.edges->exists(edge |
125.                            edge.sources->includes(node) or
126.                                edge.targets->includes(node)))
127.                    then
128.                        self.edges->forAll(edge |
129.                            edge.sources->forAll(node | self.nodes->includes(node)) and
130.                            edge.targets->forAll(node | self.nodes->includes(node)))
131.                    else false
132.                    endif
133.                else false
134.                endif
135.            end
136.        end
137.    end
```

The well formedness check (lines 120 - 137) checks that the calculation produces a single output, that all nodes are connected to at least one edge and all edges are connected to legal nodes[2]. The check is tailored to the example: in general a well formed calculation ma have more than one output.

Given a representation for expressions Exps, and a representation for calculations Calcs, the package Map is used to define how an expression is transformed into a calculation and how a calculation is checked against an expresion:

```
138.    package Mapping
139.        extends Calculations::Exps
```

2. The method uses **if then else endif** when **and** would be simpler. The reason for this is that MMT **and** is not short-circuit: both sub-expressions re evaluated. The conditional expressions are used to implement a short-circuit **and**.

The package Mapping extends Exps in order to extend each type of expression with a calculation generation method (calc) and a calculation checking method (check).

```
140.    class Number
141.        check(c:Calculations::Calcs::Calc):Boolean
142.            c.equals(self.calc())
143.        end
144.        calc():Calculations::Calcs::Calc
145.            let c = Calculations::Calcs::Calc.new(Seq{})
146.                n = Calculations::Calcs::Node.new(Seq{self.value})
147.                e = Calculations::Calcs::Edge.new(Seq{self.value.toString()})
148.            in  e.addTarget(n) []
149.                c.addNode(n) []
150.                c.addEdge(e) []
151.                c
152.            end
153.        end
154.    end
```

Class Number defines a method check (lines 141 - 143) that checks a candidate calculation to see if it represents a record of performing a constant expression. The method calc (lines 144 - 153) constructs a constant calculation.

```
155.    class Add
156.        calc():Calculations::Calcs::Calc
157.            let c1 = self.left.calc()
158.                c2 = self.right.calc()
159.            in self.addResult(c1,c2)
160.            end
161.        end
```

The class Add defines a method calc (lines 156 - 161) that constructs an addition calculation by combining the calculations for the left and right sub-expressions. The combination uses the auxiliary method addResults that adds an extra edge representing the addition operation and an extra node representing the sum:

```
162.        addResult(c1:Calculations::Calcs::Calc,
163.                  c2:Calculations::Calcs::Calc):Calculations::Calcs::Calc
164.            let result = Calculations::Calcs::Node.new(Seq{
```

```
165.                    c1.result() + c2.result()})
166.                edge = Calculations::Calcs::Edge.new(Seq{
167.                    c1.result().toString() + "+" + c2.result().toString()})
168.                calc = c1 + c2
169.            in  calc.addNode(result) []
170.                edge.addSource(c1.resultNode()) []
171.                edge.addSource(c2.resultNode()) []
172.                edge.addTarget(result) []
173.                calc.addEdge(edge) []
174.                calc
175.            end
176.        end
```

The method addResult (lines 162 - 176) merges two calculations (c1 and c2) into a single calculation that describes the addition of two sub-expressions. The two calculations are merged using '+' in line 168; the resulting calculation is extended with an edge and a node. The edge (lines 166 and 167) represents the '+' operation and the node (lines 164 and 165) represents the result. Lines 169 - 173 add the node and edge into the calculation 'calc'.

```
177.    check(c:Calculations::Calcs::Calc):Boolean
178.        c.subCalcs()->exists(c1 |
179.            if self.left.check(c1)
180.            then c.subCalcs()->exists(c2 |
181.                if self.right.check(c2)
182.                then c.equals(self.addResult(c1,c2))
183.                else false
184.                endif)
185.            else false
186.            endif)
187.    end
188. end
```

A calculation is checked against an addition expression using 'check' defines on lines 177 - 187). We could have defined check as c.equals(self.calc()). However, the given definition provides an example of *generate and test.* Using this technique we generate all the possible sub-calculations of the candidate 'c' and check them against the left and right sub-expressions of the add expression. If each sub-calculation ('c1' and 'c2') matches the appropriate sub-expression and if the result of adding the extra node and edge to 'c1 + c2' is 'c' then the check is satisfied.

The following is a simple testing method that can be used with the calculations:

```
189.    test()
190.        let  n = Calculations::Mapping::Number.new(Seq{10})
191.             add = Calculations::Mapping::Add.new(Seq{n,n})
192.        in add.check(add.calc())
193.        end
194.    end
195.end
```

The result of Calculations::test() is true thereby giving us confidence that the implementation of Add::calc is correct.

CHAPTER 7

Snapshots

- Definitions.
- Sub-snapshots.

CHAPTER 8

Relations

- Definitions of simple relations.
- Using relations as mappings.
- Using relations in both directions.
- Inheritance and relations.
- Relational combinators.

CHAPTER 9

Templates

Object-oriented systems provide modularity, polymorphism and reuse through class-based inheritance. In addition to classes, MMT provides package-based inheritance that allows systems to be constructed from groups of components.

Inheritance allows definitions to be reused systematically between groups of related components. Usually, a sub-class (or package) is intended to be partially or wholly conformant with its super-classes; in this case inheritance is used to construct *sub-type*s. Viewing an instance of a sub-class as an instance of a super-class in this way is an example of *sub-type polymorphism*.

Often an application has features which can exploit a different type of polymorphism. Consider a pair of classes that define the structure and behaviour of sequences of integers and sequences of booleans. It is not true that an instance of one class can be supplied when an instance of the second class is expected. Therefore the relationship between these classes cannot be explained using sub-typing. However the structure, operations and properties of the two classes clearly have things in common: for example taking the head of a sequence of integers involves the same activities as taking the head of a sequence of booleans; the 'length' property of both sequences is defined in exactly the same way.

The type of polymorphism that relates the classes described above is often referred to as *parametric polymorphism*. Definitions relating to classes or packages related

in this way are parameterised with respect to one or more elements. In MMT such a parameteric definition is called a *template*. A template is transformed into a concrete definition by supplying values for the parameters. In MMT this is referred to as *stamping out the template.* Unlike inheritance, the definitions resulting from stamping out a template are not related to the definitions in the template.

For example, the definitions relating to lists of things can be defined as a template that is parameteric with respect to the type of elements in the list. The concrete definitions for lists of integers and lists of booleans are then constructed by stamping out the template twice: the first time supplying Integer and the second time supplying Boolean.

Templates (parameteric polymorphism) and inheritance (sub-type polymorphism) can be used together to provide a highly expressive system description language. A typical pattern of system definition uses templates to stamp out general properties of systems as collections of packages and then uses package inheritance to combine the resulting partial descriptions to produce a single system component.

This chapter provides examples of templates.

Quote - Unquote

The language MML contains a number of constructs that either use or introduce names. For example class definitions introduce a new name that refers to a class. For example a slot reference expression uses the name of a slot.

Many components of MML expressions are sub-expressions that denote a value. When a name occurs as a sub-expression its value is found by looking the name up in the the current context. Names may also occur as components of MML expressions where the name is not intended for evaluation. Such names are referred to as *quoted*. For example, the name x is quoted in o.x, but is unquoted in x + 1. For example the name X is quoted in **class** X **end** but is unquoted in **class** Y **extends** X **end**.

By default, the following occurrence of names are quoted: package names, class names, method definition names, message names in method calls, slot names in slot reference expressions, constraint names in invariant clauses, names in attribute definitions, types, names in name space lookup expressions.

Sometimes we do not want a name to be quoted - we want the name to be computed at run-time. MML provides *unquoting brackets* that can be placed around expressions that compute names (strings) that would otherwise be quoted. The unquoting brackets are << and >>. The following package provides a somewhat artificial example of each type of unquoting:

```
1.  package <<“Example” + “Package”>>
2.      class <<“Example” + “Class”>>
3.          <<“Example” + “Attribute”>> : <<DataTypes::Integer>>;
4.          <<“ExampleMethod”>>(x:Integer):Set(<<DataTypes::Integer>>)
5.              Set{x}
6.          end
7.          m(x:Integer):Set(Integer)
8.              self.<<“Example” + “Method”>>(self.<<“ExampleAttribute”>>)
9.          end
10.         inv
11.             <<“Example” + “Constraint”>>
12.                 self.<<“Example” + “Attribute”>> > 10
13.                 fail: “Example constraint failed”
14.             end
15.     end
16. end
```

In general the expressions surrounded by unquoting brackets are not simple string concatenation. The expressions may perform arbitrary calculations in order to produce strings that are then used as names.

Unquoting brackets are particularly useful in conjunction with templates. A template definition is parameterised with respect to a number of values. The template produces a number of definitions whose names can be functions of the template parameters.

Containership

One of the simplest examples of template definitions is containership. The relation between a container and the contained elements cannot easily be expressed using inheritance. For example, Java has a number of different types of container including hashtables and vectors. Java abuses its inheritance-based type system by requir-

ing clients of containers to cast the values extracted from a container from Object to the appropriate contained element type.

Containership is essentially a definition parameterised with respect to the container and the contained element types. It can be defined as a template as follows:

```
1.  package Contains(Container:String,Contained:String)
2.      class <<Container>>
3.          <<Contained + "s">> : Set(Contains::<<Contained>>);
4.          <<"contains" + Contained>>(x:Contains::<<Contained>>):Boolean
5.              self.<<Contained + "s">>->includes(x)
6.          end
7.          <<"add" + Contained>>(x:Contains::<<Contained>>)
8.              self.<<Contained + "s">> := (self.<<Contained + "s">>->including(x))
9.          end
10.         <<"remove" + Contained>>(x:Contains::<<Contained>>)
11.             self.<<Contained + "s">> := (self.<<Contained + "s">>->excluding(x))
12.         end
13.     end
14.     class <<Contained>> end
15. end
```

Line 1 introduces a package template. A package template is defined in the same way as a package except that the name of the package is followed by a parameter list. The result of performing a package template is an object that can be used as a function: it is applied to a sequence of arguments to return a package. A package template may be applied any number of times; a new package is produced each time it is applied.

Line 1 shows that the Contains package is parameterised with respect to two strings: the name of the container class and the name of the contained elements. Lines 2 - 13 define a class. The name of the class is given by the value of the parameter supplied to the template. Normally a class name is quoted, but unquoting brackets are used to allow the name to be evaluated in the context of the body of the template.

Lien 3 defines an attribute of the container class. The name of the attribute is computed by concatenating the name of the contained element type with ‘s’. The type of the attribute is a set of elements. The type in an attribute definition is given by a type expression; in this case it is a set type expression. The element type in a set

type expression is a type expression: it must denote a type. In this case the type is the class (note **not** the class name which is the value of the variable Contained) representing the contained element type.

Lines 4 - 6 define a method for testing whether a container contains a particular element. Lines 7 - 9 define a method that is used to add a new element to a container. Lines 10 - 12 define a method that is used to remove an element from a container.

Finally line 13 defines a class named Contained. Although this class contains no definitions, it is necessary to allow the container attribute and method to refer to it as a type on lines 3, 4, 7 and 10.

Once the containership template has been defined, it can be used to produce packages by stamping it out. A template is stamped out by applying it to argument values. The following is a simple example where a library is defined to be a container of books and CDs:

```
16. package LibraryExample
17.     extends
18.         Contains("Library","Book"),
19.         Contains("Library","CD")
20.     class Book
21.         name : String;
22.     end
23.     class CD
24.         name : String;
25.     end
26. end
```

The package LibraryExample inherits from two packages resulting from stamping out the container template twice. In the first case (line 18) the container is a class named Library and the contained element is a class named Book. In the second case (line 19) the container is again Library and the contained element is is a class called CD.

The package LibraryExample also defines classes named Book and CD. By the definition of package specialisation, all definitions for Book and CD are merged together to produce the contents of LibraryExample, therefore the package definition given on lines 16 - 26 is equivalent to the following:

```
27. package LibraryExample
28.     class Library
29.         Books : Set(LibraryExample::Book);
30.         CDs : Set(LibraryExample::CD);
31.         containsBook(x:LibraryExample::Book):Boolean
32.             self.Books->includes(x)
33.         end
34.         addBook(x:LibraryExample::Book)
35.             self.Books := (self.Books->including(x))
36.         end
37.         removeBook(x:LibraryExample::Book)
38.             self.Books := (self.Books->excluding(x))
39.         end
40.         containsCD(c:LibraryExample::CD):Boolean
41.              self.CDs->includes(c)
42.         end
43.         addCD(x:LibraryExample::CD)
44.             self.CDs := (self.CDs->including(x))
45.         end
46.         removeCD(x:LibraryExample::CD)
47.             self.CDs := (self.CDs->excluding(x))
48.         end
49.     end
50.     class Book
51.         name : String;
52.     end
53.     class CD
54.         name : String;
55.     end
56. end
```

Indexed Containership

Templates can be layered on top of templates. Given a library of templates, the layering mechanism allows a wide variety of packages to be constructed simply by

supplying argument values to a layered template. This section gives a simple example of template layering.

Consider the library example of the previous section. Both books and cds are examples of contained elements that can be indexed by their names. It would be useful to add a lookup feature to a library to support simple searches.

A lookup feature based on the name of an element is a generic relationship between a named contained element and a container of indexed elements. Such a relationships may occur between the same type of container and several different types of indexed elements. Therefore, this abstract property lends itself to being represented in terms of templates as follows:

```
1.  package Named(Element:String)
2.      class <<Element>
3.          name : String;
4.          init(s:Seq(Instance)):Object
5.              self.name := (s->at(0)) []
6.              self
7.          end
8.          toString():String
9.              "<" + self.of.name + " " + self.name + ">"
10.         end
11.     end
12. end
```

Lines 1 - 12 define a template for the naming property of a contained element. Templates are not essential to represent this property: inheritance would do just as well; however it is encoded as a template for expository purposes.

```
13. package IndexedContains(Container:String,Contained:String)
14.     extends
15.         Contains(Container,Contained),
16.         Named(Contained)
17.     class <<Container>>
18.         <<"binds" + Contained>>(name:String):Boolean
19.             self.<<Contained + "s">>->exists(x | x.name = name)
20.         end
21.         <<"find" + Contained>>(name:String):IndexedContains::<<Contained>>
22.             if self.<<"binds" + Contained>>(name)
```

```
23.             then self.<<Contained + "s">>->select(x | x.name = name).selectElement()
24.             else state.error(<<Container>> + "::find" + <<Contained>> +
25.                     ": no element named " + name)
26.             endif
27.         end
28.     end
29. end
```

Lines 13 - 29 define a template called IndexedContains that is layered on the templates Contains and Named. The meaning of layering is as follows: given the name of a container and contained elements (line 13), the resulting package is a specialisation of the packages created by supplying the names to the Contains template (line 15) and the Named template (line 16).

In addition to inheriting all of the containership and naming definitions, a package resulting from stamping out IndexedContains extends the container with two methods. The method defined on lines 17 - 20 can be used as a predicate to determine whether the container binds a given name in a given binding category of containership. The method defined on lines 21 - 27 indexes a contained element of a given category given its name.

The following package shows how a library can be constructed as an indexed container of books and cds. The package also defines a simple test method for a library:

```
30. package LibraryExample
31.     extends
32.         IndexedContains("Library","Book"),
33.         IndexedContains("Library","CD")
34.     test()
35.         let b = LibraryExample::Book.new(Seq{"book1"})
36.             c = LibraryExample::CD.new(Seq{"cd1"})
37.             l = LibraryExample::Library.new(Seq{})
38.         in  l.addBook(b) []
39.             l.addCD(c) []
40.             l.findCD("cd1").toString().println()
41.         end
42.     end
43. end
```

CHAPTER 10

Graphical User Interfaces

- Classes in Gui.pkg.
- Examples of use.
- MMT tools developed using Gui classes.

CHAPTER 11

Diagrams

- How to get at diagrams.
- Exporting diagrams.
- How to define new types of diagram.
- How diagrams work.

CHAPTER 12

Meta-programming in MMT

MMT is a reflexive development environment. Many aspects of MMT are defined in MML and therefore can be inspected, extended or redefined using the techniques described in the rest of this book. The ability to be reflexive is an important factor in achieving a coherent suite of interoperable tools for developing software. A homogeneous meta-level can define and co-ordinate a wide variety of heterogenous tools, each of which is tailored to a specific aspect of an application development.

MMT aims to provide an environment within which aspect specific languages and tools can be efficiently produced and co-ordinated. The MML language consists of a suite of packages all defined in MML. These can be extended to produce an MML-like language that exhibits new features. In addition they may be instantiated to define a completely new language[1]. In all cases the basic MMT tools (written in MML) will understand the underlying representation and can interpret the new definitions. Of course, a completely new representation language will probably be best served by its own tool set; these too can be defined using MML.

This chapter provides an overview of meta-programming in MMT.

1. At the time of writing MMT has no mechanism for defining, parsing and therefore processing the concrete syntax of a non-MML-like language. Therefore such a language must be defined and used at the post-syntactic processing level, i.e. using abstract syntax trees. There are currently plans to define a parsing library in MML.

Metaclasses

Metapackages

Classifiers and Data Types

Every MMT value has a classifier. You can reach the classifier of any value by referencing the special slot named ‘of’. The classifiers of objects are instance of Class or one of its sub-classes. The classifier of all other types of value are instance of the class Classifier or one of its sub-classes.

There are three ways in which classification occurs in MMT:

1. Given a value v and a classifier c, c.checkInstance(v) return a set of strings. If the set is empty then v is classified by c. Otherwise the set contains disganostic strings describing why v is not classified by c. By default, Classifier::checkInstance runs all of the constraints defined by c in super-class to sub-class order until it reaches a constraint that fails. Classifiers may choose to redefine this method.
2. Given a value v and a classifier c, v.isKindOf(c) returns true when v is an instance of c or one of its super-classes. Instance::isKindOf does not run any ofthe constraints defined by c.
3. Given a value v, v.check() is defined as v.of.checkInstance(v).

The method Classifier::checkInstance provides a way of defining the semantics of classification via constraints. MML has a number of pre-defined constraints including:

- Object::allSlotsHaveDifferentNames that checks all the slots of an object have distinct names.
- Object::SlotsAllTypeCorrect that checks the slot values of an object are instances of the types of the corresponding attributes of its classifier. Note that the values are checked against attributes types using isKindOf and therefore will not recursively invoke checkInstance. This design decision has been taken to avoid infinite regress.

- Object::HasSlotsForAllAtts that checks an object has slots corresponding to all the attributes of its classifier.
- Class::allDifferentAttNames that checks for distinct attribute names in a class.
- Class::EmptyParents that checks a class and ensures that it has at least one parent (except for the distinguished class Instance).
- Classifier::NoCircularInheritance that checks a classifier to ensure that it does not inherit from itself.
- Classifier::ConstraintsHaveDifferentNames that checks consraints to ensure thay have distinct names.

You may add new constraints by defining sub-classes of the appropriate system class (see the chapter on meta-programming for details on extending meta-classes).

When defining classes new constraints are added to the invaviant clause of the class definition or by using Classifier::addConstraint. Classifiers of other types of values (such as sets and integer ranges) are defined as sub-types of DataTypes::DataType.

The rest of this section describes how new data types can be defined. It includes examples of classifier definitions and of parameteric classes.

Constants

Some languages permit the developer to define constants. By default MML does not have constants, however they can be defined as a new data type. This section describes how constants are defined as a sub-class of DataType and then can be used and checked as a type in attribute definitions.

The class DataType defines a method (actually inherited from Classifier) called checkInstance. A constant data type will check an instance by comparing the candidate against the single constant value. Each constant data type is different; therefore to implement constant data type a new meta-class is defined called ConstantType:

```
1.  class ConstantType extends DataTypes::DataType
2.      constant : Instance;
3.      toString():String
4.          "Const(" + self.constant.toString() + ")"
5.      end
```

```
6.      default():Instance
7.          self.constant
8.      end
9.      checkInstance(x:Instance):Set(String)
10.         if x = self.constant
11.         then Set{}
12.         else Set{x.toString() + "<>" + self.constant.toString()}
13.         endif
14.     end
15. end
```

ConstantType is a sub-class of DataType and is therefore a meta-class. An instance of ConstantType is a classifier of values. Each instance has a different constant value; therefore ConstantType defines an attribute named ‘constant’ (line 2) that is the default value of an instance (lines 6 - 8).

A ConstantType instance checks a candidate value using the method checkInstance (lines 9 - 14). The method simply compares the candidate against the constant value and returns an appropriate diagnostic.

A particular type is defined as a classifier that is an instance of ConstantType. Suppose that this is done on a constant-by-constant basis, the data type for the constant 10 is as follows:

```
16. classifier TheConstant10 metaclass ConstantType extends Instance
17.     constant = 10;
18. end
```

Line 16 defines TheConstant10 as a classifier (the **classifier** definition has the same structure as a **class** definition except that the resulting object may not define attributes). TheConstant10 is an instance of ConstantType and a sub-class of Instance. Since it is an instance of ConstantType, it has a slot named ‘constant’ that is defined to have the value 10 (line 17).

TheConstant10 may nw be used as a classifier:

```
TheConstant10.checkInstance(10) produces Set{}

TheConstant10.checkInstance(11) produces Set{11<>10}
```

TheConstant10 may now be used as a data type in class definitions:

```
19. class MiniBus
20.     passengers : Integer;
21.     maxPassengers : TheConstant10;
22.     inv
23.         CheckPassengerNumber
24.             self.passengers <= self.maxPassengers
25.             fail: "Too Full!"
26.         end
27. end
```

Instantiating ConstantType on a constant by constant basis is not particularly useful. It is much more convenient to define a classifier template that stamps out the instance each time it is used:

```
28. classifier Const(c:Instance) metaclass ConstantType extends Instance
29.   constant = c;
30.  end
```

In this case the constant value c is passed in as an argument. The definition of MiniBus becomes:

```
31. class MiniBus
32.     passengers : Integer;
33.     maxPassengers : Const(10);
34.     inv
35.         CheckPassengerNumber
36.             self.passengers <= self.maxPassengers
37.             fail: "Too Full!"
38.         end
39. end
```

Enumerated Types

An enumerated type is a collection of values; the type classifies any value in the collection. By default MML does not have enumerated types; however, they can be added by defining a meta-class called EnumeratedType and then instanting it for each new enumerated type. The class EnumeratedType is defined as follows:

```
1.  class EnumeratedType extends DataTypes::DataType
2.      elements : Set(Instance);
3.      toString():String
```

```
4.          "enum{" + self.elements->collect(e |
5.              e.toString())->asSequence.separateWith(",") + "}"
6.      end
7.      checkInstance(x:Instance):Set(String)
8.          if self.elements->includes(x)
9.          then Set{}
10.         else Set{x.toString() +
11.                 " is not an element of the enumeration " +
12.                 self.elements.toString()}
13.         endif
14.     end
15.     default():Instance
16.         self.elements.selectElement()
17.     end
18. end
```

An enumerated type has a set of values called ‘elements’ (line 2) and checks a candidate instance for inclusion in the set (lines 7 - 14). The default value for any attribute declared to have an enumerated type is an element selected at random (lines 15 - 17).

An enumerated type is created using a classifier template that is defined as follows:

```
19. classifier Enum(S:Set(Instance)) metaclass EnumeratedType extends Instance
20.     elements = S;
21. end
```

The template Enum is used to create a new type by applying it to a set of values:

Enum(Set{“male”,”female”}).checkInstance(“male”) produces Set{}

Enum(Set{“male”,”female”}).checkInstance(10) produces

Set{10 is not an element of the enumeration Set{female,male}}

Enumeration types can be used in attribute definitions:

```
22. class Person
23.     sex : Enum(“male”,”female”)
24. end
```

Tuples

A tuple is a sequence of elements, each element in the tuple may be of a different-type. A tuple type is a sequence of element types, a value classified by the tuple type is a sequence of values, each value is classified by the appropriate element type. This section shows how tuple types can be defined in terms of a meta-class named TupleType and a classifier template. Tuples of length 2 are called pairs, a template classifier is defined for constructing pair types.

```
1.  class TupleType extends DataTypes::DataType
2.      types : Seq(Classifier);
3.      toString():String
4.          "(" + self.types->collect(t | t.toString())->asSequence.separateWith(",") + ")"
5.      end
6.      checkInstance(x:Instance):Set(String)
7.          if x.isKindOf(SeqOfInstance)
8.          then if x->size = self.types->size
9.              then
10.                 let pairs = self.types.zip(x)
11.                 in pairs->iterate(p S = Set{} |
12.                     let type = p->at(0)
13.                         value = p->at(1)
14.                         checked = type.checkInstance(value)
15.                     in  if checked->isEmpty
16.                         then S
17.                         else S->including("value " + value.toString() +
18.                                           " is not of type " + type.toString())
19.                         endif
20.                     end)
21.                 end
22.             else Set{x.toString() + " does not match tuple type " + self.types.toString()}
23.             endif
24.         else Set{"tuple type " + self.types.toString() + " expects a sequence"}
25.         endif
26.     end
27.     default():Instance
28.         self.types->collect(type | type.default())->asSequence
29.     end
30. end
```

The class TupleType is defined on lines 1 - 30. The method checkInstance is defined to check the candidate is a sequence of the appropriate length and then to check each element of the sequence in turn against the element types of the tuple type.

```
31. classifier Tuple(ts:Seq(Classifier)) metaclass TupleType extends Instance
32.   types = ts;
33. end
34.
35. classifier Pair(type1:Classifier,type2:Classifier) metaclass TupleType
36.     extends Instance
37.   types = Seq{type1,type2};
38. end
```

Lines 31 - 33 define a classifier template for tuple types. Lines 35 - 38 define a classifier template for pair types.

```
Pair(Integer,Boolean).checkInstance(Seq{false,true}) produces
    Set{value false is not of type <Integer>}
Pair(Integer,Boolean).checkInstance(Seq{1,2,3}) produces
    Set{Seq{1,2,3} does not match tuple type Seq{<Integer>,<Boolean>}}
Pair(Integer,Boolean).checkInstance(1) produces
    Set{tuple type Seq{<Integer>,<Boolean>} expects a sequence}
```

Ranges

An inclusive numeric range is a pair of integers consisting of a lower bound and an upper bound. An integer is classifier by the range if it falls between the bounds or is equal to either bound. This section shows how range types are defined by a sub-class of DataType and a classifier template.

```
1.  class RangeType extends DataTypes::DataType
2.      lower:Integer;
3.      upper:Integer;
4.      toString():String
5.          "[" + self.lower + "," + self.upper + "]"
6.      end
7.      checkInstance(x:Instance):Set(String)
8.          if x.isKindOf(Integer)
```

```
9.              then
10.                 if self.lower <= x and x <= self.upper
11.                 then Set{}
12.                 else Set{x + " should be in the range [" + self.lower + "," + self.upper + "]"}
13.                  endif
14.             else Set{"range types expect an integer value"}
15.             endif
16.         end
17.         default():Instance
18.             self.lower
19.         end
20.     end
```

Lines 1 - 20 define RangeType. The checkInstance method expects an integer that falls between the lower and upper ranges.

```
21.  classifier Range(l:Integer,u:Integer) metaclass RangeType extends Instance
22.      lower = l;
23.      upper = u;
24.  end
```

Lines 21 - 24 define a classifier template for generating range types.

Range(10,20).checkInstance(15) produces Set{}

Range(10,20).checkInstance(25) produces Set{25 should be in the range [10,20]}

CHAPTER 13

The MML Grammar

This chapter defines the MML grammar. The language definition is given in EBNF using the following conventions:

- Grammar rule phrases may be grouped into a compound phrase using (and).
- Grammar rule phrases followed by * mean 0 or more repetitions.
- Grammar rule phrases enclosed in [and] are optional.
- Terminal names start with a lower case latter.
- Non-terminal names start with an upprt case letter.

The following non-terminals are assumed:

- Lparen is a left parenthesis.
- Rparen is a right parenthesis.
- Lcurl is a left curly brace.
- Rcurl is a right curly brace.
- Lsquare is a left square brace.
- Rsquare is a right square brace.
- Name is a Java identifier.
- Char is a Java character.

- Bar is a vertical bar.
- String is a Java string.
- Integer is a Java integer.

```
1.AddExp ::= MulExp (AddOp MulExp)*
2.AddOp ::= + | -
3.Apply ::= Primary [ Args ]
4.Args ::= Lparen [ Exp (, Exp)* ] Rparen
5.Attribute ::= (Name | ComputedName) : Exp [ = Exp ]
6.BoolOp ::= and | or | xor | implies
7.Boolean ::= true | false
8.BooleanExp ::= RelationalExp (BoolOp RelationalExp)*
9.Class ::=
10. class ClassName [ FunArgs ] [ Meta ] [ Supers ]
11.     [ String ]
12.     [ ClassBody ]
13.     [ Invariant ]
14. end
15.ClassBody ::= ClassBodyEntry (; ClassBodyEntry)*
16.ClassBodyEntry ::= Slot | Attribute | Method
17.ClassName ::= Name | ComputedName
18.Collection ::=  -> CollectionName
19.     [ Lparen [ Name [ Name = Exp ] Bar ] Exp (, Exp)* Rparen ]
20.CollectionName ::=
21.     append | asSequence | asSet | at | collect | exists | first | forAll |
22.         includes | including | intersection | isEmpty | iterate | last | prepend |
23.         reject | select | size | subSequence | symmetricDifference | union
24.ComputedName ::= << Exp >>
25.Cond ::= cond ([ Exp then Exp (; Exp then Exp)* ] | else Exp) end
26.Exp ::= BooleanExp (Lsquare Rsquare BooleanExp)*
27.FieldRef ::= . (Name | ComputedName) [ Args | := Apply ]
28.Fun ::= fun FunSig Exp end
29.FunArgs ::= Lparen [ Name : Exp (, Name : Exp)* ] Rparen
30.FunSig ::= FunArgs [ : Exp ]
31.If ::= if Exp then Exp [ else Exp ] endif
32.Imports ::= import Exp (, Exp)*
```

33. Invariant ::= inv InvariantEntry (; InvariantEntry)*
34. InvariantEntry ::= (Name | ComputedName | String) Exp fail: Exp
35. Let ::= (let | letrec) Name [FunArgs] = Exp in Exp end
36. Literal ::= Integer | String | Char | Boolean
37. Meta ::= metaclass Exp
38. Method ::= (Name | ComputedName) FunSig Exp
39. MulExp ::= UnaryExp (MulOp UnaryExp)*
40. MulOp ::= * | /
41. Obj ::= object Name : Exp [Slot (; Slot)*] end | @Exp [Slot (; Slot)*] end
42. Open ::= open Exp in Exp end
43. Package ::=
44. package (Name | ComputedName) [FunArgs] [Meta] [Supers] [Imports]
45. [String]
46. [PackageBody]
47. [Invariant]
48. end
49. PackageBody ::= PackageBodyEntry (; PackageBodyEntry)*
50. PackageBodyEntry ::= Package | Class | Association | Method
51. Primary ::=
52. Name | ComputedName | Literal | Set | Seq | If | Cond | Obj | Class |
53. Package | Snapshot | Let | Fun | Open | Lparen Exp Rparen
54. Ref ::= Apply (FieldRef | Collection)*
55. RelOp ::= < | > | <= | >= | <> | =
56. RelationalExp ::= AddExp RelOp AddExp
57. Seq ::= ‘Seq’ Lcurl [Exp (, Exp)*] Rcurl
58. Set ::= ‘Set’ Lcurl [Exp (, Exp)*] Rcurl
59. Slot ::= Name [(Name)] = Exp
60. Supers ::= extends Exp (, Exp)*
61. UnaryExp ::= Ref | - Ref | not Ref