The University Of Sheffield.

ReMoDeL

Object-Oriented Programming

Model Specification

Version: 0.5

Date: 20 January 2012

Anthony J H Simons
Department of Computer Science
University of Sheffield

1 Introduction

This document describes the specification for ReMoDeL OOP, a dialect of XML [1] used to encode the Object-Oriented Programming Model. This programming model is derived from the core ReMoDeL FUN functional programming model [2] and contains extensions for object-oriented programming. It also uses constructions from the ReMoDeL PKG package and namespace model [3].

1.1 Model Scope

The Object-Oriented Programming Model is intended to support a language of pure object-orientation, consisting of interfaces, classes, symbolic and basic types, fields and methods, variables, values and expressions. It should be possible to translate from ReMoDeL OOP into the strongly typed object-oriented languages, such as Java, C#, C++, Eiffel and Delphi; and also into other more radical languages, such as Smalltalk, Self and CLOS. However, it is not our intention that OOP should support every construction present in each of these languages individually. Instead, OOP captures a common subset of features that can be modelled in each of these languages.

ReMoDeL OOP is derived from the basic programming model, ReMoDeL FUN, to avoid re-inventing common constructions shared by both models, such as variables and expressions. It shares some imperative features with ReMoDeL STR, the structured programming model. The intention is to reuse, or adapt programming language constructions from other ReMoDeL XML dialects. This promotes a desirable kind of homogeneity across the different models.

1.2 Model Semantics

Since the Object-Oriented Programming Model is intended to serve as the basis for code generation in various target languages with subtly different execution semantics, a Common Semantic Model was devised, representing the semantics that could either be supported directly by, or encoded in, each target language. The major issues to resolve included: strong typing, object references, memory management, policies on inheritance, overloading, overriding and dynamic binding, class fields and methods, styles of construction and exception handling.

OOP supports strongly typed object-oriented languages. Weakly typed languages, such as Smalltalk or Objective C may also be generated, but lose the strong typing of the models. OOP class and interface types are defined within a class hierarchy, supporting a simple subtyping model (redefined method signatures have invariant argument types and covariant result types). OOP also supports generic type parameters, with optional type assumptions on the parameter. Some languages like C++ cannot check these assumptions (until parameters are bound), and weakly typed languages like Smalltalk can only render generic variables as having the most general *Object* type (but need not recover more specific type information).

Most object-oriented languages provide reference semantics by default for their object variables (a variable stores a pointer); C++ is the exception to this, supporting value semantics by default (a variable stores an object). Eiffel also has expanded types, stored by value, and C++ also offers reference and pointer variables. OOP adopts the

more common reference semantics by default for its object variables. The C++ translation requires special handling, to emulate this uniformly.

Object-oriented languages support different policies on memory management. In Smalltalk, Java, C# and Eiffel, a garbage collector is assumed to reclaim dead or unreachable objects no longer used by a program. In C++, no memory management is provided by default. OOP assumes the existence of a garbage collector. The C++ translation may either provide a collector, or must translate in such a way that objects manage their own lifetimes, for example, by use of smart pointers and reference counting.

Many object-oriented languages offer a modular mechanism for packaging sets of related classes. These may be called packages, modules, namespaces or clusters. In Java, the package represents both a namespace and a directory tree. In C++, namespaces and directories are orthogonal concepts. In Eiffel, clusters correspond to directory trees and no further partitioning by namespace is supported. OOP supports named packages with associated directory trees. Code may be generated for target languages with namespaces, or with directory trees, or both. Namespaces may be emulated in Eiffel, by adding package prefixes to class names.

Object-oriented languages offer quite diverse schemes for encapsulation at the class level. Most support a fine-grained control over visibility, offering *private* (visible locally), *protected* (visible also to inheritors) and *public* (visible to all) declarations for a class's fields and methods. Some offer *package*-level visibility (visible to others in the package) by default. Smalltalk offers a simple scheme, in which all fields are *protected* and all methods *public* by necessity. Eiffel offers a flexible export control, making different groups of methods visible to named groups of classes. OOP adopts the three-valued *private, protected* and *public* policy, with the rule that no field may be public. Eiffel may be restricted to mimic the three-value scheme; and Smalltalk can accommodate it (but will be weaker in its enforcement). At the package level, OOP supports *private* and *public* declarations, to indicate secret or exported classes and interfaces within the package.

Object-oriented languages differ in their policies on inheritance. Some, such as Smalltalk, support single inheritance (each class has one parent). Some, such as Eiffel or C++ support multiple inheritance (each class may have several parents). Some, such as Java, C# and Objective C, support a hybrid approach in which classes are defined within a single hierarchy but may also satisfy multiple interfaces, arranged in a multiple hierarchy. OOP adopts the latter position, on the grounds that more languages support this model directly, and others may emulate it, by restricting how inheritance is used. Even Smalltalk may fit this model, since objects are untyped and methods are only checked at runtime for the correct number of arguments.

Object-oriented languages differ in whether they allow name overloading within a class. Some, like Java, C# and C++, allow method names to be overloaded, such that the same name may refer to distinct methods, which are distinguished only by their type signatures. Others, like Eiffel and Smalltalk uniquely identify methods on a name-only basis. OOP adopts a no-overloading policy within a class, such that distinct names must be invented for each distinct method. Different classes may nonetheless provide different versions of the same named method.

Object-oriented languages differ in their policies on method overriding. Eiffel and Smalltalk redefine methods on a name-only basis, expecting the redefined method signature to conform to the original. C++ and C# offer the possibility of redefinition with method overriding (if the signatures conform) or overloading with method hiding (if they don't). This can result in quite different behaviour in different circumstances. OOP adopts the policy that a redefined method is intended to override. The redefined method must have unchanged argument types, but may specialise the result type.

Object-oriented languages differ in their policies on method binding. Some, like C# and C++, assume static binding by default, whereas others, like Java, assume dynamic binding by default; but both support the opposite, by explicit instruction. Other languages, like Objective C, support universal dynamic binding, whereas some, like Self, expect the binding issues to be resolved by a smart compiler. OOP adopts dynamic binding by default, so as not to restrict method redefinition, and does not require binding to be made explicit in the model. Code generators may make assumptions based on the completeness of models.

The combination of dynamic binding and overriding results in different execution semantics in different languages. In C++, Eiffel and Java, quite different behaviour may obtain when overriding a *private* method in a subclass, resulting in dynamic binding (invoking a subclass method through a superclass pointer), method hiding (removing access to a superclass method in the subclass) or a compile-time error. For this reason, OOP restricts overriding to *public* and *protected* methods. So, *private* methods may never be redefined in OOP.

Object-oriented languages differ in their support for class fields and methods (called *static*, or *shared*). While static fields are allocated once for the class, and define shared properties, static methods are more like global functions, invoked on class-objects at runtime. Not every language supports the latter concept, so OOP adopts shared fields only. Programs that require the use of static methods are in any case considered poor object-oriented style.

Object-oriented languages sometimes offer different styles of method invocation, where methods are selected according to the types of one or more arguments. OOP adopts the most prevalent style, in which one object (or expression) is the designated receiver, or target, of the method invocation and the remaining arguments are passed after resolution of the invocation. If no target object is specified, the assumed target is *self*, the current object in the enclosing method execution.

Object-oriented languages offer different styles of object creation. Some treat constructors as functional expressions that return the new object; others treat them as side-effecting methods that initialise a blank target variable. OOP treats constructors as initialising procedures (called *Creators*), but also supports functional expressions to create objects (called *Create* expressions). A *Create* expression returns an object of the specific requested type; but this object is initialised using one or more *Creators* (chained by inheritance). *Create* expressions may optionally include the target variable to initialise.

Object-oriented languages differ in their policy on exception handling. Some, like Smalltalk, have implementation-dependent mechanisms to deal with exceptions raised outside of the main flow of control. Others, like Java, C++ and C# offer a fine-

grained control over the raising and handling of exceptions. Eiffel offers a single disciplined mechanism [4], whereby a faulty method may attempt to redeem its failure. OOP adopts this approach, known as *programming by contract*, and bases exception handling on the declaration of semantic assertions about correct behaviour. This can be emulated by smart code generation in other languages.

1.3 Common Metamodel

The XML elements and attributes defined in this syntax model correspond to concepts, attributes and relationships in the ReMoDeL Metamodel. The deliberate consequence of this is that XML elements may be mapped directly onto metamodel classes. Elements are not defined in isolation, but may be organised in a conceptual hierarchy according to their similarities and differences. This is intended to support parsers that build syntax trees directly from instances of the metamodel classes, as well as parsers that use conventional XML trees.

A model will be constructed from the terminal elements in the conceptual hierarchy. A consequence of this is that certain XML element names will be reserved to denote abstract concepts in the metamodel, which are never actually present in any instance of the model. These abstract elements are nonetheless defined as part of the model, since they may correspond to strongly typed nodes in syntax trees derived directly from the metamodel.

The intention is for all ReMoDeL XML dialects to be mapped onto a common metamodel. The terminal elements used across different languages, though they may be different, will nonetheless share certain similarities, expressed through their relationships with common abstract elements.

1.4 XML Conventions

ReMoDeL OOP adopts all the W3C conventions for XML [1]. Identifier symbols must observe the rules of Unicode identifiers, attribute values must be enclosed in quotes and special characters must be escaped as entity references. Apart from this, ReMoDeL imposes a certain "house style" on identifiers:

- The names of all XML elements are presented in *CapitalCase*, similar to type names in the Java programming language.

- The names of all XML attributes are presented in *camelCase*, similar to variable names in the Java programming language.

The use of hyphens and underscores as part of identifier names goes against the house style and is strongly discouraged. The use of digits as part of the body of an identifier is legal, but also generally discouraged, unless the application clearly demands this.

2 Types

The kinds of *Types* used in ReMoDeL OOP subdivide into the *Basic* types, the *Symbolic* types, the *Class* types and the *Interface* types. There is also a *Generic* type parameter. These all descend from the metaclass *Type*, as illustrated in the fragment of the ReMoDeL Metamodel shown in figure 1. The metaclass *Record* is not needed in OOP, since it is superseded by *Class*. Likewise the metaclass *Array* is not needed, since OOP uses predefined library collection classes instead.

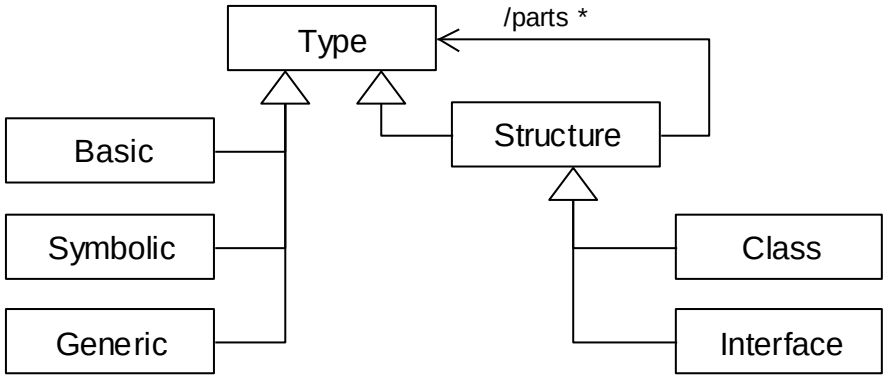

Figure 1: Types in the ReMoDeL Metamodel

All types are defined as instances of one of these metaclasses, which also specify the type's implementation. For example, an *Integer* type is defined as an instance of the *Basic* metaclass; the symbolic *Boolean* type is defined as an instance of the *Symbolic* metaclass; a *Person* class is defined as an instance of the *Class* metaclass; and the *Iterator* interface is defined as an instance of the *Interface* metaclass. Instances of the *Generic* metaclass are type parameters, placeholders for types.

ReMoDeL assumes standard default specifications for the types: *Boolean*, *Character*, *Integer*, *Natural*, *Decimal* and *Void*, which may be taken for granted if no alternative definition is provided. *Boolean* occupies a byte, *Character* supports UTF-8, *Integer* assumes a signed 32-bit range, *Natural* assumes an unsigned 32-bit range, *Decimal* assumes double-precision floating point and *Void* is the most specific type of the empty value *null*. These types are declared in the *Core* package for the ReMoDeL OOP dialect, which has other basic types, such as *Byte*, *Short* and *Long*, and also the class *String*, which may be included in models as desired.

2.1 Basic Type

The *Basic* element is used to declare a primitive value-type with a fixed range of values. Basic types are often built-in types of the target language. The syntax supports declaring types with different ranges (and therefore different precision), which must be preserved by the implementation of the target language.

The grammar for the *Basic* element is:

```
<!ELEMENT Basic (Literal, Literal)>
<!ATTLIST Basic name NMTOKEN #REQUIRED>
```

The attributes of the *Basic* element are:

- *name* – the name of the type (required).

The type name must be given in *CapitalCase* (see section 1.4).

The children of the *Basic* element are exactly one each of:

- *Literal* – the lower limit of the type (optional);
- *Literal* – the upper limit of the type (optional);

Examples of the *Basic* element include:

```
<Basic name="Character"/>

<Basic name="Integer">
  <Literal value="-2147483648" limit="low" type="Integer"/>
  <Literal value="2147483647" limit="high" type="Integer"/>
</Basic>
```

In the first example, the range of the declared *Character* type is assumed to be the default range supporting UTF-8. The second example explicitly declares the *Integer* type to have a 32-bit range.

2.2 Symbolic Type

The *Symbolic* element is used to declare a symbolic value-type with enumerable values. These are symbolic constants, which are declared by finite enumeration. Symbolic types are often user-defined, but may be built-in types.

The grammar for the *Symbolic* element is:

```
<!ELEMENT Symbolic (Literal+)>
<!ATTLIST Symbolic name NMTOKEN #REQUIRED>
```

The attributes of the *Symbolic* element are:

- *name* – the name of the type (required).

The type name must be given in *CapitalCase* (see section 1.4).

The children of the *Symbolic* element are one or more ordered occurrences of:

- *Literal* – a symbolic constant of the type (required, one-to-many).

Examples of the *Symbolic* element include:

```
<Symbolic name="Boolean" from="Core">
  <Literal value="false" type="Boolean"/>
  <Literal value="true" type="Boolean"/>
</Symbolic>
```

```
<Symbolic name="Status">
  <Literal value="closed" type="Status"/>
  <Literal value="open" type="Status"/>
  <Literal value="frozen" type="Status"/>
</Symbolic>
```

Model transformation tools must be able to determine whether any given symbolic type is primitive (built-in), or requires special declaration in each target language.

2.3 Generic Type

The *Generic* element is used to declare generic parameters. These are placeholders for actual types. Generic parameters are included as the first children inside any generic class or method definition.

The grammar for the *Generic* element is:

```
<!ELEMENT Generic EMPTY>
<!ATTLIST Generic name NMTOKEN #REQUIRED>
<!ATTLIST Generic inherit NMTOKEN #IMPLIED>
<!ATTLIST Generic satisfy NMTOKEN #IMPLIED>
```

The attributes of the *Generic* element are:

- *name* – the name of the parameter (required);
- *inherit* – an upper bound class name (optional);
- *satisfy* – an upper bound interface name (optional).

Both must be given in *CapitalCase* (see section 1.4). By default, any type may be substituted later for a generic parameter. If an upper bound class is supplied, only classes that inherit from this class may be substituted. If an upper bound interface is supplied, only types that satisfy this interface may be substituted. At most one of the optional attributes is used.

The *Generic* element has no children.

Examples of the *Generic* element include:

```
<Generic name="Element"/>

<Generic name="Element" satisfy="Compare"/>

<Generic name="Human" inherit="Person"/>
```

The first example declares a type parameter for the element type of a general list or tree. The second example is a parameter for the element type of a sorted list or tree, which must satisfy the interface type *Compare*, since any type replacing the parameter must possess comparison operations. The third example declares a type parameter for any type, which inherits from the *Person* class, since it must possess at least the methods defined in *Person*.

2.4 Class Type

The *Class* element is used to declare class types. These are named structures, consisting of named fields, each of a different type, and named methods, describing the operations of the class. Class types may declare generic type parameters for the types of one or more of their fields or methods. Class types may also assert invariant properties for the class as a whole.

The grammar for the *Class* element is:

```
<!ELEMENT Class (Generic*, Inherit?, Satisfy*, Employ*, Assert*,
    Field*, Creator*, Method+)>
<!ATTLIST Class name NMTOKEN #REQUIRED>
<!ATTLIST Class visible (private|public) "public">
<!ATTLIST Class abstract (false|true) "false">
<!ATTLIST Class library (false|true) "false">
```

The attributes of the *Class* element are:

- *name* – the name of the class (required);
- *visible* – the package export status of the class (optional);
- *abstract* – true, if the class is abstract (optional);
- *library* – true, if the class is in a predefined library (optional).

Class names must be given in *CapitalCase* (see section 1.4). The package export status of a class is usually *public* (the default), but may be made *private* (not exported from the package). A class may optionally be declared *abstract* (if it has at least one abstract method), and may belong to a predefined *library* (in which case it only contains signatures, rather than full definitions).

The children of the *Class* element are:

- *Generic* – a generic type parameter (zero-to-many);
- *Inherit* – refers to the immediate superclass (optional);
- *Satisfy* – refers to any satisfied interface (zero-to-many);
- *Employ* – refers to any other employed type (zero-to-many);
- *Assert* – the class invariant assertion (optional);
- *Field* – a named field member (zero-to-many);
- *Creator* – a named constructor member (zero-to-many);
- *Method* – a named method member (one-to-many).

The only compulsory child to have is at least one *Method*. A class should declare at least one *Creator* if it also defines *Fields*. If no *Inherit* reference is supplied, the default superclass is understood to be *Object*, the root of the class hierarchy.

Examples of the *Class* element include:

```
<Class name="Person" visible="public">
  <Assert contract="age within range" when="always">
    <!-- assertion body omitted -->
  </Assert>
  <Field name="forename" type="String" visible="private"/>
  <Field name="surname" type="String" visible="private"/>
  <Field name="gender" type="Character" visible="private"/>
  <Field name="age" type="Natural" visible="private"/>
  <Creator name="make" type="Void" visible="public">
    <Variable name="forename" type="String"/>
    <Variable name="surname" type="String"/>
    <Variable name="gender" type="Character"/>
    <Variable name="age" type="Natural"/>
    <!-- creator body omitted -->
  </Creator>
  <Method name="getForename" type="String" visible="public">
    <!-- method body omitted -->
  </Method>
  <Method name="getSurname" type="String" visible="public">
    <!-- method body omitted -->
  </Method>
  <Method name="getGender" type="Character" visible="public">
    <!-- method body omitted -->
  </Method>
  <Method name="getAge" type="Natural" visible="public">
    <!-- method body omitted -->
  </Method>
</Class>
```

This example is of a user-defined *Person* class, with *Fields* and *Methods* of the given names and types. It is exported from its owning package (not shown). It asserts a class invariant, whose contract will ensure that the value of the *age Field* is always within a suitable range. It has a single *Creator*, with formal arguments whose names deliberately alias the *Field* names – this is typical in OOP, since the scope of identifiers may be resolved explicitly. The *Methods* are designed to return the values stored in the respective *Fields*, and are appropriately typed. Since these are access methods, they have no formal arguments.

```
<Class name="Pair" visible="public" library="true">
  <Generic name="First"/>
  <Generic name="Second"/>
  <Field name="first" type="First" visible="private"/>
  <Field name="second" type="Second" visible="private"/>
  <Creator name="make" type="Void" visible="public">
    <Variable name="first" type="First"/>
    <Variable name="second" type="Second"/>
  </Creator>
  <Method name="getFirst" type="First" visible="public"/>
  <Method name="getSecond" type="Second" visible="public"/>
</Class>
```

This example is of a generic *Pair* class type, the building block for arbitrary pairs in OOP. It has two generic parameters, *First* and *Second*, standing for two arbitrary types, which are used to type the first and second projections. Because this is a predefined library class, no method- or creator-bodies are required.

A *Class* may be defined to inherit from a parent *Class*, known as its superclass. In this case, the *Inherit* reference dependency is used (see the ReMoDeL Package and Namespace Model [3]). An example of this is:

```
<Class name="Student" visible="public">
  <Inherit refer="Person" kind="Class" from="People"/>
  <Field name="registration" type="Natural" visible="private"/>
  <Creator name="make" override="true"
        type="Void" visible="public">
    <Variable name="forename" type="String"/>
    <Variable name="surname" type="String"/>
    <Variable name="gender" type="Character"/>
    <Variable name="age" type="Natural"/>
    <Variable name="registration" type="Natural"/>
    <!-- creator body omitted -->
  </Creator>
  <Method name="getRegistration" type="Natural" visible="public">
    <!-- method body omitted -->
  </Method>
</Class>
```

This defines the *Student* class to be a subclass of *Person*, adding the extra *Field* called *registration*. In ReMoDeL OOP, at most one superclass may be specified; and if no superclass is specified, the default superclass is *Object*, the root class from the *Core* package. The class *Person* is here assumed to exist in a foreign *Package* called *People*, which must have been previously consulted by the current *Package*, for the name to be in scope.

Inheritance also affects how objects are created. The *Student* class must define how to initialise its own instances; here it redefines the *Creator* called *make* (see section 4.4), so that this accepts more arguments than the inherited version. The new version will call the inherited version, to initialise inherited fields.

A *Class* may also be defined to satisfy an *Interface*, in which case the *Satisfy* reference dependency is used. The satisfying class must provide concrete methods for each signature declared in the interface. An example is where the *Account* class satisfies the interface *Asset*:

```
<Class name="Account" visible="public" abstract="true">
  <Satisfy refer="Asset" kind="Interface" from="Finance"/>
  <Employ refer="Status" kind="Symbolic" from="Finance"/>
  <Employ refer="Person" kind="Class" from="People"/>
  <!-- class body omitted for brevity -->
</Class>
```

We assume that the *Asset* interface defined a signature for *getValue*, an abstract method, which class *Account* now implements, perhaps by returning the account's current balance. In ReMoDeL OOP, a class may satisfy as many interfaces as is desired, so long as it implements all of the declared abstract methods (or, declares itself to be an *abstract* class). Note in this example how the *Employ* reference

dependency is also used to express the dependency on other types declared in this, and other, packages. In general, all structured types should explicitly declare their fine-grained dependencies on other types. Certain types, such as the standard types from the *Core* package, may be assumed to exist automatically.

2.5 Interface Type

The *Interface* element is used to declare interface types. These are named structures, consisting of named methods, describing the abstract operations of the interface. Interface types may declare generic type parameters for the types of one or more of their methods. Interface types may also assert invariant properties for the interface as a whole.

The grammar for the *Interface* element is:

```
<!ELEMENT Interface (Generic*, Satisfy*, Employ*, Assert*,
    Field*, Method+)>
<!ATTLIST Class name NMTOKEN #REQUIRED>
<!ATTLIST Class visible (private|public) #FIXED "public">
<!ATTLIST Class abstract (false|true) #FIXED "true">
<!ATTLIST Class library (false|true) "false">
```

The attributes of the *Interface* element are:

- *name* – the name of the interface (required);
- *visible* – always public, since an interface must be exported;
- *abstract* – always true, since an interface is always abstract;
- *library* – true, if the interface is in a predefined library (optional).

Interface names must be given in *CapitalCase* (see section 1.4). The package export status of an interface is always *public*, since otherwise there would be no point in declaring it. An interface is always *abstract* (since its methods are all abstract), and may belong to a predefined *library*.

The children of the *Interface* element are:

- *Generic* – a generic type parameter (zero-to-many);
- *Satisfy* – refers to any super-interfaces (zero-to-many);
- *Employ* – refers to any other employed type (zero-to-many);
- *Assert* – the interface invariant assertion (optional);
- *Field* – a shared field constant (zero-to-many);
- *Method* – a named method signature (one-to-many).

The only compulsory child to have is at least one *Method* signature. An interface may optionally declare *Fields*, if these are shared, initialised constants. If no *Satisfy*

reference is supplied, the default super-interface is understood to be *Anything*, the root of the interface hierarchy.

An example of the *Interface* element is the following:

```
<Interface name="Equal" visible="public"
          abstract="true" library="true">
  <Generic name="Type"/>
  <Method name="equals" type="Boolean" visible="true">
    <Variable name="other" type="Type"/>
  </Method>
  <Method name="notEquals" type="Boolean" visible="true">
    <Variable name="other" type="Type"/>
  </Method>
</Interface>
```

This is the definition of the *Equals[Type]* interface declared in the *Core* standard library. It is generic in the type parameter *Type*, since classes satisfying this interface will supply methods comparing like *Type* with *Type*. Any class implementing the two methods specified here will satisfy this interface. An interface may also optionally declare *Fields*, if these are shared, public, initialised constants.

An Interface may be defined to satisfy a super-interface, in which case the *Satisfy* reference dependency is used (see the ReMoDeL Package and Namespace Model [3]).

```
<Interface name="Compare" visible="public"
          abstract="true" library="true">
  <Generic name="Type"/>
  <Satisfy refer="Equal[Type]" kind="Interface" from="Core"/>
  <Method name="lessThan" type="Boolean" visible="true">
    <Variable name="other" type="Type"/>
  </Method>
  <Method name="moreThan" type="Boolean" visible="true">
    <Variable name="other" type="Type"/>
  </Method>
  <Method name="noLessThan" type="Boolean" visible="true">
    <Variable name="other" type="Type"/>
  </Method>
  <Method name="noMoreThan" type="Boolean" visible="true">
    <Variable name="other" type="Type"/>
  </Method>
</Interface>
```

This is the definition of the *Compare[Type]* interface declared in the *Core* standard library. It is generic in the type parameter *Type*, since classes satisfying this interface will supply methods comparing like *Type* with *Type*. Notice how this interface in turn satisfies the super-interface *Equal[Type]*, which defines equality and non-equality as methods. Any class implementing the six comparison methods illustrated will satisfy this interface.

An *Interface* may satisfy as many other interfaces as desired. If it declares no super-interface, then the implicit super-interface is *Interface*, the root interface.

3 Expressions

The kinds of *Expression* used in ReMoDeL OOP subdivide into the simple *Literal*, *Identifier*, and *Branch* expressions; the evaluating expressions *Assign, Operator*, *Create* and *Invoke*; the compound expressions *Sequence, Parallel, Select* and *Iterate* and the control statements *Assert, Rescue* and *Return*. These all descend from the metaclass *Expression*, as illustrated in the fragment of the ReMoDeL Metamodel shown in figure 2.

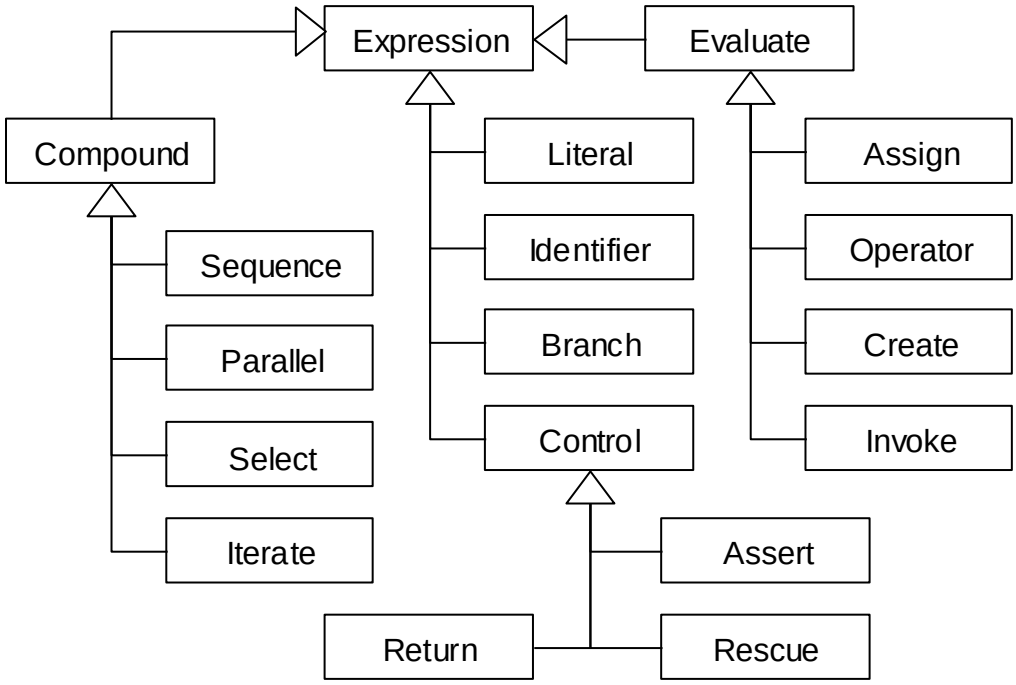

Figure 2: Expressions in the ReMoDeL Metamodel

All expressions are defined as instances of one of these metaclasses. For example, the literal values *3* or *"hello"* are instances of the *Literal* metaclass; an occurrence of the variable *count* is defined as an instance of the *Identifier* metaclass; a mathematical operation is an instance of the *Operator* metaclass; object construction is an instance of the *Create* metaclass and a method invocation is an instance of the *Invoke* metaclass. A sequential block of statements is an instance of the *Sequence* metaclass. Conditionally executed statements are instances of the *Select* metaclass, denoting branching. The *Assert* and *Rescue* metaclasses define exception handling, and *Return* indicates returned values. All expressions have a *type* attribute (even if *Void*).

3.1 Literal Expression

The *Literal* element may be used to define a literal expression, or constant. Simple literal values may be declared for any type whose values have a directly printable representation. Literals of structured types may not be declared in OOP, which requires the use of object construction using *Create* instead. Literal elements may state whether they are the low or high limit for their type.

The grammar for the *Literal* element is:

```
<!ELEMENT Literal (Literal | Identifier)*>
<!ATTLIST Literal value CDATA #IMPLIED>
<!ATTLIST Literal limit (low | high) #IMPLIED>
<!ATTLIST Literal type NMTOKEN #REQUIRED>
```

The attributes of the *Literal* element are:

- *value* – the literal value, a constant (optional; required in OOP);
- *limit* – whether this value is the low or high limit (optional);
- *type* – the type of the literal value (required).

Any *value* that has a printed representation is valid, including integral and floating-point numbers, single and multiple character strings or symbols. The *value* is always supplied as text data. The interpretation of the *value* is given by the corresponding *type*, which is the defined name, or inferred name, of any declared ReMoDeL OOP type (see section 2).

The *Literal* element may not have children in OOP; although it may in other ReMoDeL dialects. In OOP, *Literal* instances always represent instances of basic or symbolic types, or character strings, which are instances of the *String* class. Instances of structured types should instead be created using the *Create* object construction expression.

Simple examples of the *Literal* element include:

```
<Literal value="false" type="Boolean"/>

<Literal value="e" type="Character"/>

<Literal value="-10" type="Integer"/>

<Literal value="42" type="Natural"/>

<Literal value="3.1415926" type="Decimal"/>

<Literal value="null" type="Void"/>
```

These examples show how to express literals of simple types. The value *null* is the only legal value of the *Void* type. Examples of *Literal* elements used to define the upper and lower limits of a *Basic* type include:

```
<Literal value="-2147483648" limit="low" type="Integer"/>

<Literal value="2147483647" limit="high" type="Integer"/>

<Literal value="0" limit="low" type="Natural"/>

<Literal value="4294967295" limit="high" type="Natural"/>
```

The first two examples define the limits of a signed 32-bit Integer type. The second two examples define the limits of an unsigned 32-bit Natural type. Refer to *Basic Types* for more examples (see section 2.1).

3.2 Identifier Expression

The *Identifier* element may be used to define an occurrence of a variable that was previously declared, and which is currently in scope.

The grammar for the *Identifier* element in OOP is:

```
<!ELEMENT Identifier EMPTY)>
<!ATTLIST Identifier name NMTOKEN #REQUIRED>
<!ATTLIST Identifier type NMTOKEN #REQUIRED>
<!ATTLIST Identifier state (before | after) #IMPLIED>
<!ATTLIST Identifier scope (local | object | special) "local">
```

The attributes of the *Identifier* element are:

- *name* – the name of the identifier (required);
- *type* – the type of the identifier (required);
- *state* – the prior, or posterior state (optional):
- *scope* – the scope of the identifier (optional).

The *name* must be supplied in *camelCase*, and the *type* must be a legal ReMoDeL OOP type name (see section 2). The *state* attribute is only used inside postconditions, taking the value *before* or *after*. The *scope* attribute is used to specify the scope of an identifier. In ReMoDeL OOP, the allowed scopes are *local* (the default), *object* and *special*. An identifier with *local* scope refers to a local variable, or method argument currently in scope. An identifier with *object* scope refers to an object field. An identifier with *special* scope refers to one of the three special reserved identifiers: *self*, *super* or *result*, which are used in particular contexts. Since OOP identifiers are always scoped, it is custom to use the same names for both *Fields* and local *Variables*, which simplifies naming conventions in OOP.

The *Identifier* element has no children.

Examples of the *Identifier* element include:

```
<Identifier name="age" type="Natural"/>

<Identifier name="age" type="Natural" scope="object"/>

<Identifier name="self" type="Person" scope="special"/>
```

The first example is of an identifier referring (by default) to a locally scoped *Variable* called *age* of the type *Natural* (this could be a method argument, or locally declared variable). The second example is of an identifier referring unambiguously to an object's *Field* of the same name and type. The third example shows how to refer to the *self*-referential variable, standing for the current object.

3.3 Operator Expression

The *Operator* element may be used to define an operator expression. This is any standard operation using the predefined mathematical, Boolean or comparison operator symbols supported by ReMoDeL OOP. All such symbols have standard ReMoDeL names, which are mapped to the appropriate symbol by model transformation tools.

The grammar for the *Operator* element is:

```
<!ELEMENT Operator ((Literal | Identifier | Operator |
          Create | Invoke),
    (Literal | Identifier | Operator | Create | Invoke)?)>
<!ATTLIST Operator symbol (not | or | and | implies |
    equals | notEquals | lessThan | moreThan |
    noMoreThan | noLessThan | negate | plus | minus |
    times | divide | modulo) #REQUIRED>
<!ATTLIST Operator type NMTOKEN #REQUIRED>
```

The attributes of the *Operator* element are:

- *symbol* – the standard name of the operator symbol (required);
- *type* – the result type of the operator expression (required).

The *type* must be any legal ReMoDeL OOP type name (see section 2). The predefined legal *symbol* names to be used include the following (where *noMoreThan* denotes less than or equal, and *noLessThan* denotes greater than or equal):

- logical operators: *not, or, and* and *implies*;
- comparison operators: *equals, notEquals, lessThan, moreThan, noMoreThan* and *noLessThan*;
- arithmetic operators: *negate, plus, minus, times, divide* and *modulo*.

The logical and comparison operators yield a *Boolean* valued result. The arithmetic operators are polymorphic, returning some *Number* type consistent with the argument types, such as *Integer, Natural* or *Decimal*.

The children of the *Operator* element are its operands, which include one or more occurrences of:

- any expression that returns a value of a suitable type (required).

The majority of the predefined operators are binary and expect two operands. The operators *not* and *negate* are unary operators and only expect a single operand.

Examples of the *Operator* element include:

```
<Operator symbol="negate" type="Integer">
  <Literal value="42" type="Integer"/>
</Operator>
```

```
<Operator symbol="plus" type="Integer">
  <Literal value="7" type="Integer"/>
  <Literal value="35" type="Integer"/>
</Operator>

<Operator symbol="lessThan" type="Boolean">
  <Literal value="7" type="Integer"/>
  <Operator symbol="minus" type="Integer">
    <Literal value="12" type="Integer"/>
    <Literal value="6" type="Integer"/>
  </Operator>
</Operator>
```

The first example is of a unary operation, negating the value of the operand 42. The second example is of a simple binary operation, adding two integer operands. The third example is of a Boolean comparison of two values, where the second value is a nested operation.

3.4 Assign Expression

The *Assign* element may be used to define a variable re-assignment. This includes assigning a new value to a variable and also modifying the in-place contents of a variable. All side-effecting updates are performed with the *Assign* element, which is available in those ReMoDeL dialects that support side-effecting updates. The different kinds of assignment operator have standard ReMoDeL names, which are mapped to the appropriate symbol by model transformation tools.

The grammar for the *Assign* element is:

```
<!ELEMENT Assign (Identifier, (Literal | Identifier |
    Operator | Assign | Create | Invoke)?)>
<!ATTLIST Assign symbol
    (equals | plus | minus | times | divide) #REQUIRED>
<!ATTLIST Assign type NMTOKEN #FIXED "Void" >
```

The attributes of the *Assign* element are:

- *symbol* – the standard name of the operator symbol (required);
- *type* – the *Void* type (required).

The *type* of an assignment is always *Void*. The most common assignment operator symbol is *equals*, indicating that the target will be set equal to the second operand. The legal *symbol* names to be used include the following:

- binary re-assignment: *equals;*
- binary in-place update: *plus, minus, times* and *divide;*
- unary in-place update: *plus* and *minus*;

Assignment always returns the type *Void*, since it is bad practice to embed assignments inside functional expressions.

The children of the *Assign* element are its operands, which include:

- an *Identifier* expression, denoting the variable to update (required);
- any compatible expression to be assigned (required; or not needed).

Standard re-assignment is a binary operation, requiring both an identifier and a value expression. In-place update may be a unary or binary operation – if unary, the implicit second operand is always the unit value 1.

Examples of the *Assign* element include:

```
<Assign symbol="equals" type="Void">
  <Identifier name="counter" type="Integer"/>
  <Literal value="35" type="Integer"/>
</Assign>

<Assign symbol="plus" type="Void">
  <Identifier name="counter" type="Integer"/>
</Assign>
```

The first example assigns the value 35 to the variable *counter*. The second example increments the value of this *counter* by one. *Assign* is similar to, but distinct from *Operator*, because of its side-effecting behaviour. Translation tools may insert special processing tasks, such as reasserting invariants, when an assignment is detected.

3.5 Create Expression

The *Create* element may be used to define an object creation expression. This is an evaluating expression that allocates a new object of a specific type and invokes that object's *Creator* procedure to initialise it. The *Create* element refers to the chosen initialisation procedure by name, which must be defined for the target class. It may include a target variable to initialise as its first child, or, if this is absent, returns the created object.

The grammar for the *Create* element is:

```
<!ELEMENT Create (Identifier?, (Literal | Identifier |
    Operator | Create | Invoke)*)>
<!ATTLIST Create creator NMTOKEN #REQUIRED>
<!ATTLIST Create implicit (false | true) "false">
<!ATTLIST Create type NMTOKEN #REQUIRED>
```

The attributes of the *Create* element are:

- *creator* – the name of the creator procedure (required);
- *implicit* – true, if the target of creation is omitted (optional);
- *type* – the result type of the created object (required).

The *creator* name must be given in *camelCase*, and should refer to a previously defined *Creator* for the object being created. The *type* must be the class name of the object being created, in *CapitalCase*. This class should have a *Creator* of the given

name. If *implicit* is set to true, this means that the target variable to initialise is implicit, and the result of creation will be passed as an initial value to the next enclosing expression, some kind of *Variable* or *Field* declaration.

The children of the *Create* element include an optional identifier to initialise, and zero or more construction arguments as further expressions:

- the target identifier to initialise (optional, alternates with *implicit*);
- further argument-expressions (zero-to-many).

The argument-expressions must evaluate to suitable values, whose types correspond to the types expected by the creator procedure's arguments. It is common for a class to provide one creator, conventionally called *create*, that expects no arguments. Other creators are uniquely named (see section 1.2) and expect arguments.

Examples of the *Create* element include:

```
<Create creator="create" type="Person">
  <Identifier name="person" type="Person"/>
</Create>

<Variable name="person" type="Person">
  <Create creator="create" implicit="true" type="Person"/>
</Variable>
```

The first example creates an instance of *Person* (determined by the value of the *type* attribute), and invokes the creator called *create* to initialise this object, storing the result in the target identifier *person*, which refers to a previously declared variable. By convention, the default creator (with no arguments) is always given the name *create*. This is the creation style to use if the target variable is already declared.

The second example illustrates the use of the *implicit* attribute, indicating the omission of the target variable. In this case, the result of creation is passed as an initial value to the surrounding *Variable* expression. This is the only case where *Create* needs no children. This is the creation style to use if the target variable is being initialised as part of its declaration.

```
<Create creator="make" type="Person">
  <Identifier name="person" type="Person"/>
  <Literal value="John" type="String"/>
  <Literal value="Smith" type="String"/>
  <Literal value="m" type="Character"/>
  <Literal value="24" type="Natural"/>
</Create>

<Create creator="make" type="Pair[String, Natural]">
  <Identifier name="person" type="Pair[String, Natural]"/>
</Create>
```

The third example illustrates a more common style of creation, in which both the target variable and a set of construction arguments are supplied as children of the *Create* element. The construction arguments have the types *String*, *String*, *Character*,

Natural; this assumes that the creator called *make* in the class *Person* was defined to expect four arguments of these types.

The last example shows how to create an instance of a generic class, *Pair*. The type attribute declares how the class's two type parameters *First*, *Second* (see section 2.4) are to be instantiated with the actual types *String* and *Natural*. In all these examples, the same type was used for the *Create* expression and the target *Variable* receiving the created object. In general, the *Variable* may have a more general type than the created object. This is determined from the class hierarchy

3.6 Invoke Expression

The *Invoke* element may be used to define a method invocation expression. This is an evaluating expression that invokes a named *Method* on a target object, which is typically the first child expression. The *Method* element refers to the chosen method by name, which must be defined for the class of the target object. It may include a target object expression as its first child, or, if this is absent, assumes that the target object is *self*, the currently executing object.

The grammar for the *Invoke* element is:

```
<!ELEMENT Invoke ((Literal | Identifier | Operator |
          Create | Invoke)?,
    (Literal | Identifier | Operator | Create |
          Invoke)*)>
<!ATTLIST Invoke method NMTOKEN #REQUIRED>
<!ATTLIST Invoke implicit (false | true) "false">
<!ATTLIST Invoke type NMTOKEN #REQUIRED>
```

The attributes of the *Invoke* element are:

- *method* – the name of the method to invoke (required);
- *implicit* – true, if the target object is omitted (optional);
- *type* – the result type of the invoked method (required).

The *method* name must be given in *camelCase*, and should refer to a previously defined *Method* for the class of the target object. The *type* must be the result type of this method, in *CapitalCase*. If *implicit* is set to true, this means that the target object is implicit, and is understood to be *self*, the current object.

The children of the *Invoke* element include an optional target object expression, and zero or more method arguments as further expressions:

- the target object expression (optional, alternates with *implicit*);
- further argument-expressions (zero-to-many).

The argument-expressions must evaluate to suitable values, whose types correspond to the types expected by the method's arguments. It is possible for some methods to expect no arguments, if they access field values in the target object. All methods are uniquely named within a given class (see section 1.2).

Examples of the *Invoke* element include:

```
<Invoke method="getAge" type="Natural"/>
  <Identifier name="person" type="Person"/>
</Invoke>

<Invoke method="getAge" implicit="true" type="Natural"/>

<Invoke method="getAge" type="Natural"/>
  <Identifier name="self" scope="special" type="Person"/>
</Invoke>
```

The first example invokes the no-argument access method *getAge* on a target variable *person,* whose type is the class *Person*. A *Method* with this name must be defined for the class *Person*. The target object expression need not be an identifier, but can be any expression that yields an object when it is evaluated.

The second example illustrates the use of the *implicit* attribute, indicating the omission of the target object expression. In this case, the target object is implicitly understood to be *self*, the currently executing object, here assumed to be a *Person*. Methods may explicitly refer to *self*, a special identifier (see section 3.2), usually when passing the current object as an argument. The third example refers explicitly to *self*, and is equivalent to the second example in meaning.

```
<Invoke method="setAge" type="Void">
  <Identifier name="person" type="Person"/>
  <Literal value="24" type="Natural"/>
</Invoke>

<Invoke method="setAge" implicit="true" type="Void">
  <Literal value="24" type="Natural"/>
</Invoke>
```

In general, the *Invoke* element will require one or more additional method argument expressions as its children. The fourth example invokes the method *setAge* on the object denoted by the *person* identifier (the first sub-expression) and supplies a literal expression as a method argument (the second sub-expression). The fifth example invokes the same method implicitly on *self*. In this case, the *Invoke* element need only contain the argument expression as its child. *Invoke* should supply as many argument expressions as the named *Method* expects, and the expressions should be of suitable compatible types.

3.7 Branch Expression

The *Branch* element may be used to define a lazy-evaluating expression. The *Branch* element is a kind of *Expression*, representing a guard or trigger protecting the enclosed expression, whose evaluation is delayed. In OOP, *Branch* is used in conjunction with *Select*, a generalised conditional branching construction, and *Iterate*, a generalised deterministic and conditional looping construction. When a selection choice-expression is evaluated, and the result matches the trigger for a particular branch, this identifies which branch of the program to follow.

The grammar for the *Branch* element in OOP is:

```
<!Element Branch (Assign | Return | Create | Invoke |
    Select | Iterate | Sequence | Parallel)>
<!ATTLIST Branch when CDATA #REQUIRED>
<!ATTLIST Branch type NMTOKEN #REQUIRED>
```

The attributes of the *Branch* element are:

- *when* – the trigger value for the branch (required);
- *type* – the type of the delayed expression (required).

The trigger *when* is a value determining when the branch should be executed. This can be a *Boolean* value (*false, true*), a scalar value (a symbolic or integral value) or the result of any other choice-expression tested by *Select* or *Iterate* expressions (both available in OOP, but not in all ReMoDeL dialects). The *type* may be any legal ReMoDeL type name (see section 2), but must correspond to the type of the wrapped expression, whose evaluation is delayed.

The children of the *Branch* element are:

- the expression whose evaluation is delayed.

Boolean-triggered examples of the *Branch* element include:

```
<Branch when="true" type="String">
  <!-- branch expression omitted -->
</Branch>

<Branch when="false" type="String">
  <!-- branch expression omitted -->
</Branch>
```

Scalar-triggered examples of the *Branch* element include:

```
<Branch when="closed" type="String">
  <!-- branch expression omitted -->
</Branch>

<Branch when="open" type="String">
  <!-- branch expression omitted -->
</Branch>

<Branch when="frozen" type="String">
  <!-- branch expression omitted -->
</Branch>
```

These examples are the cases for a multi-branch selection on some expression of the type *Status*, each wrapping an expression, here a *String* value to be returned. Cases of other scalar types are possible, such as *Integer, Natural* or *Character*, so long as all cases are mutually exclusive and exhaustive. There is no default case.

3.8 Select Expression

The *Select* element may be used to define a conditional branching expression. It is a generalisation of the binary branching if-statement and the multi-branching switch-

statement (or case-statement) in programming languages. The *Select* expression is one of four *Compound* expressions available in OOP (the others being: *Iterate*, *Sequence*, and *Parallel*). *Select* is used in combination with lazy-evaluating *Branch* expressions in OOP.

The grammar for the *Select* element in OOP is:

```
<!ELEMENT Select ((Literal | Identifier | Operator |
    Create | Invoke), Branch+)>
<!ATTLIST Select choice (simple | multiple) #REQUIRED>
<!ATTLIST Select type NMTOKEN #REQUIRED>
```

The attributes of the *Select* element are:

- *choice* – whether the choice is *simple* or *multiple* (required);
- *type* – the result type of the selection expression (required).

The *choice* indicates whether a simple or multiple-choice selection is intended. The *type* may be any legal ReMoDeL OOP type name (see section 2), but must either be identical to, or a supertype of, the expected result type of each *Branch*. Formally, the type returned by *Select* is the least upper bound of the types returned by all of its branches. For single-branch *Select*, the result type must be *Void*.

The *Select* element has one choice-*Expression* child and one or more *Branch* children (in OOP, single-branch *Select* is permitted):

- the choice-expression, having a *Boolean* type for a simple choice and a scalar type for a multiple choice (required);
- the branch expressions, of which one may be taken (required, one-to-many).

The first child is always the choice-expression. This has a *Boolean* value in binary selections (*if-then-else* statement), or any scalar value in a multi-branching selection (*switch, case* statement).

Examples of the *Select* element include:

```
<Select choice="simple" type="Void">
  <Operator symbol="equals" type="Boolean">
    <Identifier name="count" type="Natural"/>
    <Literal value="0" type="Natural"/>
  </Operator>
  <Branch when="true" type="Void">
    <Assign symbol="plus" type="Void">
      <Identifier name="count" type="Natural">
   </Assign>
  </Branch>
</Select>
```

This first example is a simple selection with a *Boolean*-valued choice expression. In OOP, simple selections may have one or two branches. This example is of a single-branching if-statement, whose optional branch is selected when *count* = 0. A single-branching *Select* cannot return any value, since the *false* case does not exist (and so

cannot return a value). The result type must be *Void* for both branches. Compare the above with the next example:

```
<Select choice="simple" type="String">
  <Operator symbol="moreThan" type="Boolean">
    <Identifier name="count" type="Natural"/>
    <Literal value="0" type="Natural"/>
  </Operator>
  <Branch when="true" type="String">
    <Return type="String">
      <Literal value="More than zero" type="String">
    </Return>
  </Branch>
  <Branch when="false" type="String">
    <Return type="String">
      <Literal value="Equal to zero" type="String">
    </Return>
  </Branch>
</Select>
```

This second example is another simple selection, but this time with two branches, each of which returns a value. *Select* must always return the least upper bound type (the most specific type that is a supertype of each of the branch types). An individual *Branch* may return a more specific type than the result type declared for *Select*. In this case, both branches return a *String*, and the *Select* therefore returns a *String*.

```
<Select choice="multiple" type="String">
  <Identifier name="status" type="Status"/>
  <Branch when="closed" type="String">
    <Return type="String">
      <Literal value="Account is closed" type="String"/>
    </Return>
  </Branch>
  <Branch when="open" type="String">
    <Return type="String">
      <Literal value="Account is open" type="String"/>
    </Return>
  </Branch>
  <Branch when="frozen" type="String">
    <Return type="String">
      <Literal value="Account is frozen" type="String"/>
    </Return>
  </Branch>
</Select>
```

This third example is a multi-branching selection, with a scalar-valued choice-expression of the enumerated type *Status*. In OOP, multi-branching expressions must exhaustively cover all possible branches (missing branches are not permitted). So, it is useful to be able to use finite *Symbolic* types as the values of guards. (Code generators are free to generate *default* branches that raise an exception).

3.9 Iterate Expression

The *Iterate* element may be used to define a deterministic, or conditional looping expression. It is a generalisation of the deterministic for-loop and the conditional while-loop. The *Iterate* expression is one of four *Compound* expressions available in

OOP (the others being: *Select*, *Sequence*, and *Parallel*). *Iterate* is used in combination with lazy-evaluating *Branch* expressions in OOP.

The grammar for the *Iterate* element in OOP is:

```
<!ELEMENT Iterate (Variable | (Literal | Identifier |
    Operator | Create | Invoke), Branch)>
<!ATTLIST Iterate loop (bounded | dynamic) #REQUIRED>
<!ATTLIST Iterate type NMTOKEN #FIXED "Void">
```

The attributes of the *Iterate* element are:

- *loop* – whether the loop is *bounded* or *dynamic* (required);
- *type* – the result type of the *Iterate* expression (must be *Void*).

The value of *loop* indicates what kind of iteration is desired: *bounded* indicates a deterministic loop over a known range, *dynamic* indicates a conditional loop with an unknown number of iterations. The *type* of an *Iterate* expression is always *Void*, since the repeated *Branch* is always executed for its side effects.

The *Iterate* element has two children, of which the first must be chosen according to the type of loop desired:

- the loop control expression, a control *Variable* of any suitable type for a bounded loop, or any *Boolean* expression for a dynamic loop (required);
- a *Branch* expression, wrapping the body of the iteration (required).

If the loop is *bounded*, the first child must be a *Variable* declaration, containing the details of the range of the iteration. If the loop is *dynamic*, the first child may be any *Boolean* valued entry condition to the loop.

Examples of the *Iterate* element include:

```
<Iterate loop="bounded" type="Void">
  <Variable name="count" type="Integer" step="plus">
    <Literal value="0" type="Integer"/>
    <Literal value="10" type="Integer"/>
  </Variable>
  <Branch when="next" type="Void">
    <!-- loop body omitted -->
  </Branch>
</Iterate>

<Iterate loop="bounded" type="Void">
  <Variable name="count" type="Integer" step="minus">
    <Literal value="10" type="Integer"/>
    <Literal value="0" type="Integer"/>
  </Variable>
  <Branch when="next" type="Void">
    <!-- loop body omitted -->
  </Branch>
</Iterate>
```

These are both examples of bounded iteration that repeat ten times. The first example initialises the control variable *count* to 0, and increments this after each cycle, halting when the value of *count* reaches 10. That is, the loop body is executed once for each of the ascending values 0..9 inclusive, but halts at 10. The second example initialises *count* to 10, and decrements this after each cycle, halting when the value of *count* reaches 0. That is, the loop body is executed once for each of the descending values 10..1 inclusive, but halts at 0. The loop body may refer to the value of *count* through an *Identifier* expression also called *count*.

Control variables have a *step* attribute that is set to one of *plus* or *minus*, when performing this kind of counted iteration, depending on whether it is desired to visit elements in ascending, or descending order. The *Variable* element always contains two children: an initialisation expression, and a backstop expression, denoting when to halt. The backstop is always "one past the end" of the range of the iteration. Another kind of bounded iteration in OOP supports visiting every element of a collection:

```
<Iterate loop="bounded" type="Void">
  <Variable name="member" type="Person" step="next">
    <Identifier name="membership" type="List[Person]"/>
  </Variable>
  <Branch when="next" type="Void">
    <!-- loop body omitted -->
  </Branch>
</Iterate>
```

In this case, the *step* attribute is set to *next*, meaning that the variable should range over each element of the following collection. Here, the *member* variable will range over every *Person* object in the following expression, which has the collection type *List[Person]*. When the *Variable*'s *step* attribute is set to *next*, it only expects one child element, any collection expression, and the variable is initialised to the first element of the collection, and is reset to the next element on each cycle, until no elements are left. The loop body may refer to the value of *member* through an *Identifier* of the same name. However, the collection should not be updated while iteration is in progress (this behaviour is undefined).

For *bounded* iterations, the *Branch* is triggered on the special condition *next*, which only holds when the next value of the control variable is available. On each cycle, the control variable will take on the next value, until no more values are available, whereupon the condition fails to hold, so the loop terminates. For *dynamic* iterations, the *Branch* is triggered when the *Boolean* control expression is *true*:

```
<Iterate loop="dynamic" type="Void">
  <Operator symbol="lessThan" type="Boolean">
    <Identifier name="count" type="Integer"/>
    <Literal value="10" type="Integer"/>
  </Operator>
  <Branch when="true" type="Void">
    <!-- loop body omitted -->
  </Branch>
</Iterate>
```

In this example, a *Boolean*-valued loop control expression is tested before each cycle; and the loop body is executed only if the outcome is *true*. The loop will iterate for as long as this condition remains *true*, and will halt if it becomes *false*.

3.10 Sequence Expression

The *Sequence* element may be used to define a compound block of statements, which are executed in sequential order. It is one of four *Compound* expressions available in OOP (the others being: *Select*, *Iterate* and *Parallel*). A *Sequence* is used as the top-level expression in the body of a *Method* or *Creator*. A *Sequence* may also introduce local *Variable* declarations at the start of the block, or return a value at the end of the block.

The grammar for the *Sequence* element in OOP is:

```
<!ELEMENT Sequence (Variable*, (Assign | Create |
    Invoke | Select | Iterate | Parallel)*, Return?)>
<!ATTLIST Sequence type NMTOKEN #REQUIRED>
```

The attributes of the *Sequence* element are:

- *type* – the result type of the *Sequence* expression (mandatory).

The type of a *Sequence* is usually *Void*, but may be some other type, if the last statement in the *Sequence* is a *Return* expression, in which case the Sequence must be given the same type as the *Return* expression.

The children of the *Sequence* element include:

- local *Variable* declarations (zero to many);
- various statement expressions (zero to many);
- a *Return* expression (optional).

If the *Sequence* declares local *Variables* at the start of the block, the rest of the block may contain *Identifier* expressions having the same names, through which the values of the variables may be accessed or modified. Local *Variables* may be bound to values as they are declared; they become unbound at the end of the block.

Examples of the *Sequence* element include:

```
<Sequence type="Void"/>

<Sequence type="Void">
  <Invoke method="push" type="Void">
    <Identifier name="stack" type="Stack[Integer]"/>
    <Literal value="42" type="Integer"/>
  </Invoke>
  <Invoke method="pop" type="Void">
    <Identifier name="stack" type="Stack[Integer]"/>
  </Invoke>
</Sequence>
```

The first example is of an empty Sequence, commonly used as the body of a method that deliberately does nothing, a null operation. The second example shows a Sequence consisting of two ordered statements, method invocations that first *push*, then *pop* a *stack*. The statements are executed in the order that they appear.

In the second example, the *stack Identifier* is assumed to correspond to a local *Variable* that is already in scope. Two more examples below illustrate how a *Sequence* may also declare local variables at the head of the sequence:

```
<Sequence type="Void">
  <Variable name="number" type="Integer"/>
  <Assign symbol="equals" type="Void">
    <Identifier name="number" type="Integer"/>
    <Literal value="42" type="Integer"/>
  </Assign>
</Sequence>

<Sequence type="Integer">
  <Variable name="number" type="Integer">
    <Literal value="42" type="Integer"/>
  </Variable>
  <Assign symbol="equals" type="Void">
    <Identifier name="number" type="Integer"/>
    <Literal value="7" type="Integer"/>
  </Assign>
  <Return type="Integer">
    <Identifier name="number" type="Integer"/>
  </Return>
</Sequence>
```

The third example declares a *Variable number*, and later assigns the value of 42 to this. The fourth example declares the *Variable number* and initialises it to the value of 42, and later re-assigns the value 7 to it. Several *Variables* may be declared and initialised at the head of a *Sequence*. *Variables* may not be introduced partway through a *Sequence*.

The third example executes the statements in the *Sequence* for their side effects, and the *Sequence* returns no result. By contrast, the fourth example illustrates how a *Return* expression may be placed as the last in a *Sequence*, indicating that it does return a result, in which case the *Sequence* must have the same type as the *Return* expression.

3.11 Parallel Expression

The *Parallel* element may be used to define a compound block of statements, which are all executed in parallel. It is one of four *Compound* expressions available in OOP (the others being: *Select*, *Iterate* and *Sequence*). A *Parallel* element introduces a concurrent fork, whose separate statements each execute in a separate parallel thread; but all the threads re-synchronise at the end of the *Parallel* element, which is a concurrent join. A *Parallel* block may also introduce local *Variable* declarations at the start of the block. These declarations represent variables that are shared among the parallel threads, to which access is automatically synchronised.

The grammar for the *Parallel* element in OOP is:

```
<!ELEMENT Parallel (Variable*, (Assign | Create |
    Invoke | Select | Iterate | Sequence)*)>
<!ATTLIST Parallel type NMTOKEN #FIXED "Void">
```

The attributes of the *Parallel* element are:

- *type* – the result type of the *Parallel* expression (must be Void).

The type of a *Parallel* expression must be *Void*, since the competing parallel threads cannot return values in any orderly fashion. Communication with the rest of the program is through side effects on variables.

The children of the *Parallel* element include:

- local *Variable* declarations (zero to many);
- concurrent thread expressions (zero to many).

If the *Parallel* element declares local *Variables* at the start of the block, the concurrent threads in the rest of the block may contain *Identifier* expressions having the same names, through which the values of the variables may be accessed or modified. Access to these local *Variables* is automatically synchronised, that is, the threads may read and write to them independently, without fear of corrupting their state. Local *Variables* may be bound to initial values as they are declared; they become unbound at the end of the block.

Examples of the *Parallel* element in OOP include:

```
<Parallel type="Void">
  <Variable name="account" type="Account">
    <Invoke method="getAccount" type="Account">
      <Identifier name="bank" type="Bank"/>
      <Literal value="3002032" type="Natural"/>
    </Invoke>
  </Variable>
  <Invoke method="credit" type="Void">
    <Identifier name="account" type="Account"/>
    <Literal value="200" type="Integer"/>
  </Invoke>
  <Invoke method="debit" type="Void">
    <Identifier name="account" type="Account"/>
    <Literal value="50" type="Integer"/>
  </Invoke>
</Parallel>
```

In this example, a local *Variable account* is initialised to an *Account* instance obtained from a *Bank*, and the same *account* is credited and debited, in no particular order. Because the *account* is a synchronised variable, one or other of these threads will block the other, until it releases the *account*. Synchronisation is enforced at the level of the expression accepting the variable as its immediate child.

So in this case, there is not a great advantage in the parallel encoding – the effect is to allow the credit and debit to happen in an arbitrary order. Compare the above with the following example:

```
<Parallel type="Void">
  <Invoke method="credit" type="Void">
    <Identifier name="savings" type="Account"/>
    <Literal value="200" type="Integer"/>
  </Invoke>
  <Invoke method="debit" type="Void">
    <Identifier name="current" type="Account"/>
    <Literal value="50" type="Integer"/>
  </Invoke>
</Parallel>
```

In this example, parallel threads are started to credit the *Account savings*, while simultaneously debit the *Account current*. These operations are conceptually concurrent and may literally overlap, that is, execute simultaneously, although the implementation of multi-threading will depend on the target architecture.

3.12 Return Statement

The *Return* element is a kind of *Control* statement that may be used to return a value from a method. *Return* encloses the value to be returned as the result of a method. Explicit *Return* expressions are needed in OOP and other imperative programming models, in which statements are evaluated for their side effects.

The grammar for the *Return* element is:

```
<!ELEMENT Return (Literal | Identifier | Operator |
    Assign | Create | Invoke)>
<!ATTLIST Return type NMTOKEN #REQUIRED>
```

The attributes of the *Return* element are:

- *type* – the type of the enclosed expression (required).

The *type* may be any legal ReMoDeL OOP type name (see section 2), and must be the same as the type of the enclosed expression, whose value is being returned.

The child of a *Return* element is:

- any simple expression with a non-*Void* type

Return elements may only be found in two locations, as determined by the grammar. A *Return* element can be:

- the very last element in a *Sequence;* or
- the immediate child of a *Branch* in a *Select.*

If a *Sequence* contains a *Return* expression, the *Sequence* must declare the same type as the *Return* expression. If a *Branch* contains a *Return*, then every *Branch* of the same *Select* expression must contain a *Return*, and must have a type that is compatible with the type declared by the *Select* expression.

3.13 Assert Statement

The *Assert* element is a kind of *Control* statement that may be used to define a semantic assertion that is checked at runtime, and which raises an exception if the assertion is violated. *Assert* may be used to specify a method precondition, or a method postcondition, or a data type invariant. Exceptions may later be handled by *Rescue* blocks. The behaviour of *Assert* and *Rescue* follows the programming by contract metaphor from the Eiffel programming language.

The grammar for the *Assert* element is:

```
<!ELEMENT Assert (Literal | Identifier | Operator | Invoke)>
<!ATTLIST Assert contract CDATA #REQUIRED>
<!ATTLIST Assert when (always | before | after) #REQUIRED>
<!ATTLIST Assert type NMTOKEN #FIXED "Void" >
```

The attributes of the *Assert* element are:

- *contract* – the name of the asserted contract (required);
- *when* – whether to assert *before, after* or *always* (required);
- *type* – this is always *Void* (required).

The value of *contract* is a short descriptive string, labelling the property being asserted. This always names the positive property asserted, rather than the negative exceptional condition caused by breaking the property. The value of *when* is an enumerated value from the set *{before, after, always}* denoting whether the assertion is respectively a precondition, postcondition or datatype invariant. The *type* of an assertion is always *Void*.

Assert elements may be found as children of *Class* and *Interface* elements, expressing datatype invariants. They may also be found as children of *Method* and *Creator* elements, expressing preconditions or postconditions.

The child of an *Assert* element is:

- any truth-valued expression with a *Boolean* type.

This can be any literal, identifier, operator expression or method invocation that asserts a truth-valued property. The asserted expression may refer to local variables, object fields or formal arguments that are in scope. If the assertion is a postcondition, it may reason about the prior and posterior states of variables, whose optional *state* attribute is set to one of *before* (prior state) or *after* (posterior state). Otherwise, all variables are presumed not to have changed state. A postcondition may refer to the special identifier *result*, referring to the result returned by a method.

Examples of the *Assert* element include:

```
<Class name="Person">
  <Assert contract="valid lifespan" when="always" type="Void">
    <Operator symbol="noMoreThan" type="Boolean">
      <Identifier name="age" scope="object" type="Natural"/>
      <Literal value="120" type="Natural"/>
    </Operator>
  </Assert>
  <Field name="age" type="Natural" visible="private"/>
  <!-- class contents omitted for brevity -->
</Class>
```

This declares a class invariant for the class *Person*, stating that the *age* attribute, which is a *Natural* number, can never exceed 120, the maximum expected human lifespan.

```
<Method name="deposit" type="Void" visible="public">
  <Variable name="amount" type="Integer"/>
  <Assert contract="positive amount" when="before" type="Void">
    <Operator symbol="moreThan" type="Boolean">
      <Identifier name="amount" type="Integer"/>
      <Literal value="0" type="Integer"/>
    </Operator>
  </Assert>
  <Assert contract="balance increased" when="after" type="Void">
    <Operator symbol="equals" type="Boolean">
      <Identifier name="balance" state="after"
            scope="object" type="Integer"/>
      <Operator symbol="plus" type="Integer">
        <Identifier name="amount" type="Integer"/>
        <Identifier name="balance" state="before"
          scope="object" type="Integer"/>
      </Operator>
    </Operator>
  </Assert>
  <!-- method body omitted for brevity -->
</Method>
```

This example declares a method *deposit*, which has a precondition to ensure that the deposited *amount* is positive; and has a postcondition to ensure that the *balance* increased exactly by the *amount* deposited. Notice how the postcondition uses the optional *state* attribute of an *Identifier* to describe the *balance* in both its prior and posterior states, since this variable changes state during the method's execution.

```
<Method name="squareRoot" type="Natural">
  <Variable name="number" type="Natural"/>
  <Assert contract="result is root" when="after" type="Void">
    <Operator symbol="equals" type="Boolean">
      <Identifier name="result" scope="special" type="Natural"/>
      <Operator symbol="times" type="Natural">
        <Identifier name="number" type="Natural"/>
        <Identifier name="number" type="Natural"/>
      </Operator>
    </Operator>
  </Assert>
  <!-- method body omitted for brevity -->
</Method>
```

This example declares a postcondition using the special identifier *result* to refer to the result of the method. The assertion ensures that squaring the result is equal to the

original number, whose square root was taken. Notice how there is no need to use the *state* attribute in this example, since no variables change state during the execution of the method.

3.14 Rescue Statement

The *Rescue* element is a kind of *Control* statement that may be used to define a handler for certain exceptions raised in a method. The *Rescue* block sits outside the main *Sequence* of a method body and wraps one or more remedial statements, whose goal is to restore an object to a clean state. The *Rescue* element may allow a failed *Method* to be subsequently reattempted.

The grammar for the *Rescue* element is:

```
<!ELEMENT Rescue (Assign | Create | Invoke |
    Select | Iterate | Sequence | Parallel)*>
<!ATTLIST Rescue restart CDATA #IMPLIED>
<!ATTLIST Rescue type NMTOKEN #FIXED "Void">
```

The attributes of the *Rescue* element are:

- *restart* – the number of restart attempts (optional);
- *type* – this is always *Void* (required).

The value of *restart*, if supplied, is a natural number, representing the number of times the rescued method may be restarted, before the handler gives up and passes control to the next exception handler.

Exception handling in ReMoDeL OOP follows Eiffel's programming by contract metaphor [4], in which methods take appropriate responsibility for dealing with failure. If a method's precondition is broken, the method is not directly responsible for this, but raises an exception in the caller. If after being invoked with valid arguments the method breaks its own postcondition, or its owning class's datatype invariant, the method is held responsible, so an exception is raised locally. If the method contains a *Rescue* block, it may clean up after the failure, to restore the object to a stable state. It may then either pass the failure up the stack, or attempt to restart, in the hope of success.

The child of a *Rescue* element is:

- any simple or compound expression, typically with a *Void* type.

This can be any statement, but is typically a *Sequence* of remedial statements. Notice how the *Rescue* block cannot return a value; this is because its purpose is to restore the object to a stable state. It does not replace the action carried out by the method. The only way for a method to return is if it succeeds normally, possibly after a restart.

Translations into languages like Java and C++ that throw exception objects must ensure that the generated code obeys the programming by contract rule. Preconditions should be tested outside any protected *try*-block for that method; postconditions and invariants should be tested inside the protected *try*-block. If an assertion fails, this

should *throw* a *BrokenContract* exception. The catch-handler should expect to deal with this class of exception. If a restart is possible, the whole method must be wrapped in a loop, so that it may be reattempted. If no restarts are left, the handler must raise the same *BrokenContract* exception again.

Examples of the *Rescue* element include:

```
<Rescue restart="1" type="Void">
  <Assign symbol="equals" type="Void">
    <Identifier name="balance" type="Integer" scope="object"/>
    <Operator symbol="plus" type="Integer">
      <Identifier name="balance" type="Integer" scope="object"/>
      <Identifier name="amount" type="Integer"/>
    </Operator>
  </Assign>
</Rescue>
```

This *Rescue* block rolls back the effect of a failed *withdraw* transaction. We may assume that the *balance* fell below some minimum value, such that the current *withdraw* transaction was refused, because it broke a postcondition. The contained expression is an assignment, which restores the *balance* to its previous value, by adding the *amount* that was previously deducted. Since *restart* has the value 1, this indicates that the method may be reattempted once – but in all likelihood it will fail again, in which case the same exception will be raised, and the handler will clean up again, before passing the exception up the call stack.

The *Rescue* block may refer to all variables that are in scope, including the method's formal arguments and the object's fields. It cannot refer to local variables in the method's body, which are out of scope. If a restart is permitted, then the method will execute with the same values for its arguments (which may not be changed by the remedial actions). Therefore, any cleaning up may only really affect the object's state, through its fields and other reachable objects.

ReMoDeL OOP adopts a particular view on exception handling. It is deliberately not as flexible as languages like Java, C# or C++, which support the throwing and catching of arbitrary typed exceptions, which may be intercepted by type. This tends to support "programming control flow by exceptions," which we wish to avoid. Instead, the notion is that methods should always seek to behave correctly, but if they cannot, they must fail in a disciplined way. Translations into languages with richer exception frameworks may assume that all violations of assertions that are recoverable will raise the exception called *BrokenContract*. Other violations of system hardware and the like will raise the exception called *SystemError*.

4 Declarations

The kinds of *Declaration* used in ReMoDeL OOP include all *Class*, *Interface*, *Method*, *Creator*, *Field* and *Variable* declarations. Significant kinds of *Declaration* include *Classifier*, which defines all things with a namespace; and *Property*, which defines named properties with an associated *Type*. Descendants of *Property* include *Member*, an owned property with an associated visibility, which contrasts with *Variable*, a free property that is not a member of a structured type. Figure 2 illustrates a fragment of the ReMoDeL Metamodel dealing with declarations.

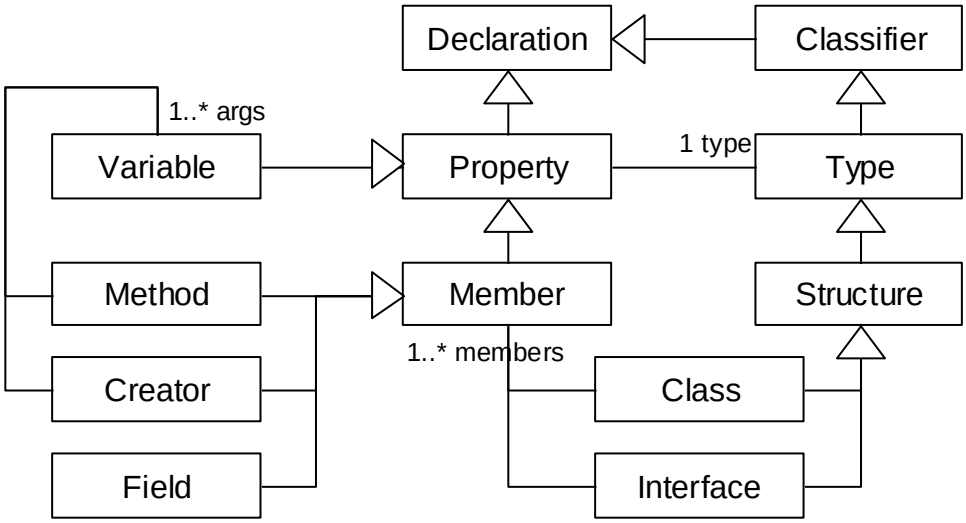

Figure 3: Declarations in the ReMoDeL Metamodel

Program declarations are instances of these metaclasses. For example, class fields, constructors and methods are respectively instances of the *Field*, *Creator* and *Method* metaclasses; and any global or local variable definition is an instance of the *Variable* metaclass. All the *Class*, *Interface*, *Basic*, *Symbolic* and *Generic* types of the language considered above (see section 2) are also kinds of *Declaration*.

4.1 Variable Declaration

The *Variable* element may be used to define named storage for program values. *Variable* is used to declare both global and local variables, and also formal arguments to methods. A *Variable* is a kind of *Property* (from which it obtains a property name and associated typed storage). In all ReMoDeL dialects supporting imperative styles of programming, *Variable* also plays an important role in controlling iteration.

The grammar for the *Variable* element in OOP is:

```
<!ELEMENT Variable ((Literal | Identifier | Operator |
          Create | Invoke)?,
    (Literal | Identifier | Operator | Create | Invoke)?)>
<!ATTLIST Variable name NMTOKEN #REQUIRED>
<!ATTLIST Variable step (plus | minus | next) #IMPLIED>
<!ATTLIST Variable type NMTOKEN #REQUIRED>
```

The attributes of the *Variable* element are:

- *name* – the name of the variable (required);
- *step* – the incremental step (optional; used in iteration);
- *type* – the type of the variable (required).

The *name* must be supplied in *camelCase*, the expected style for all property names. The *step* relates to how a variable may be reset automatically during iteration (see section 3.9). If the *Variable* is used to control iteration, *step* must be set to one of the enumerated values: *plus*, meaning increment by one; *minus*, meaning decrement by one; or *next*, meaning iterate over the next element of a collection. The *type* may be any legal ReMoDeL OOP type name (see section 2).

The *Variable* element optionally has the children:

- any initial value to bind to the variable (optional);
- a suitable backstop value delimiting a range (optional).

Initial value expressions may be used to initialise a local *Variable* at the point of introduction. Initialising a local *Variable* is usually more convenient than first declaring it and then assigning a value to it using *Assign* with a target *Identifier* having the same variable name. Local *Variable* expressions are initialised in the order they are declared, at the head of the owning *Sequence*.

Backstop expressions are only used during iteration, to declare a limit to the range of values, over which the variable may iterate. The initial value is always included in the iteration; whereas the backstop value is always excluded from the iteration – it is always "one past the end" of the iteration.

Simple examples of the *Variable* element include:

```
<Variable name="amount" type="Integer"/>

<Variable name="primes" type="List[Natural]"/>
```

These examples show how to declare variables of both simple and structured types. This is the style to use when declaring formal method arguments (see section 4.3), as well as local variables. Further examples below show how a local *Variable* may also be initialised at the point of declaration:

```
<Variable name="rate" type="Decimal">
  <Literal value="3.25" type="Decimal"/>
</Variable>

<Variable name="answer" type="Integer">
  <Operator symbol="times" type="Integer"/>
    <Literal value="7" type="Integer"/>
    <Literal value="6" type="Integer"/>
  </Operator>
</Variable>
```

```
<Variable name="pundit" type="Person">
  <Create creator="make" implicit="true" type="Person">
    <Literal value="Ronald" type="String"/>
    <Literal value="Manager" type="String/>
  </Create>
</Variable>
```

These examples show how variables may be initialised to any general expression. The first initialises *rate* with a literal value 3.25; the second initialises *answer* to the product of 7 times 6; the third initialises *pundit* to the result of object creation.

The following examples show the usage of the *Variable* element during iteration. Declaring a *step* attribute is only legal during iteration. The *step* controls how the *Variable* is initialised and re-initialised on each iteration.

```
<Variable name="member" type="Person" step="next">
  <Identifier name="membership" type="List[Person]"/>
</Variable>
```

In this first example, the *Variable member* is initialised to the first element of the collection of *Person* objects stored in the variable *membership*, but will iterate over each element in the list, until there are no more. This kind of iteration is indicated by setting *step* to the value *next*, indicating general iteration. A more specific kind of counted iteration is indicated by setting the *step* to *plus* or *minus*:

```
<Variable name="count" type="Integer" step="plus">
  <Literal value="0" type="Integer"/>
  <Literal value="10" type="Integer"/>
</Variable>

<Variable name="count" type="Integer" step="minus">
  <Literal value="10" type="Integer"/>
  <Literal value="0" type="Integer"/>
</Variable>
```

These examples show the use of the backstop value. The *Variable count* is initialised to the initialisation expression, and iteration will continue until the backstop is reached. The *step* value *plus* indicates that the variable will increment by 1; whereas the *step* value *minus* indicates that the variable will decrement by 1 on each iteration. Counted iteration always covers a contiguous range and cannot jump in steps larger than 1. For examples of the context in which these expressions are used during iteration, see above (see section 3.9).

4.2 Field Declaration

The *Field* element may be used to define a named field of a *Class* or *Interface*. A *Field* is a kind of *Member* (from which it obtains a *visibility*) and transitively a kind of *Property* (from which it obtains a name and a type). A *Field* may be declared with an initial default value. If used in an *Interface*, a *Field* may only declare a shared constant value.

The grammar for the *Field* element in OOP is:

```
<!ELEMENT Field (Literal | Identifier | Operator |
          Create | Invoke)?>
<!ATTLIST Field name NMTOKEN #REQUIRED>
<!ATTLIST Field shared (false | true) "false">
<!ATTLIST Field visible (private | protected | public) "private">
<!ATTLIST Field type NMTOKEN #REQUIRED>
```

The attributes of the *Field* element are:

- *name* – the name of the field (required);
- *shared* – the allocation of the field (optional);
- *visible* – the visibility of the field (optional);
- *type* – the type of the field (required).

The *name* must be supplied in *camelCase*, the expected style for all property names. The optional *shared* attribute, when *true*, indicates whether the *Field* is shared by all instances of the declaring class (also known as *static* in Java and C++). By default, *shared* is *false*, meaning that the *Field* is replicated in each instance. The *visible* attribute may be set to any of: *private* (visible in the declaring class), *protected* (also visible in descendant classes) or *public* (visible to all). By default, a *Field* will be *private*. The *type* may be any legal ReMoDeL FUN type name (see section 2).

The *Field* element optionally has the child:

- any initial value to bind to the field (optional).

Initial value expressions may be used to declare and initialise a *Field* at the point of introduction. In ReMoDeL OOP, the initial value is treated as the default value for the *Field*, which may later be replaced during object construction. However, a *shared* field may only be initialised in this way. If an *Interface* is ever given a *Field*, this must be declared *shared* and have an initial value; the *Field* is then treated as a shared constant.

Simple examples of the *Field* element include:

```
<Field name=”forename” type=”String” visible=”private”/>

<Field name=”age” type=”Natural” visible=”protected”/>
```

These show class *Fields* of different types, and with different visibility declarations. In general, *private* visibility offers the strictest access control, allowing only methods of the owning *Class* to access the *Field*. Where a *Class* is part of a hierarchy, granting the *Field protected* visibility will allow subclasses to access the *Field* also.

The following examples show how the above *Fields* may be given default values, at the point of introduction:

```
<Field name=”forename” type=”String” visible=”private”>
  <Literal value="John" type="String"/>
</Field>
```

```
<Field name=”age” type=”Natural” visible=”protected”>
  <Literal value="0" type="Natural"/>
</Field>
```

Assuming that these *Fields* are owned by the *Class* called *Person*, then any *Person* object created without an explicit *forename* will have the name "*John*"; and any object created without an explicit *age* will have the age 0. Translators will deal with this in different ways, for example Java supports both default and constructed *Field* values, whereas in C++, default values may be supplied using the constructor's initialisation syntax, and explicit values may be replaced using assignment in the body.

The following example illustrates a shared *Field*, allocated once for a *Class* called *SavingsAccount*, which is also initialised at the point of declaration:

```
<Field name=”netRate” type=”Decimal” shared=”true”>
  <Operator symbol=”minus”>
    <Invoke method="getGrossRate" type="Decimal">
      <Identifier name="rates"
          scope="object" type="InterestRates"/>
    </Invoke>
    <Invoke method="getTaxRate" type="Decimal">
      <Identifier name="rates"
          scope="object" type="InterestRates"/>
    </Invoke>
  </Operator>
</Field>
```

This shows how *netRate* is declared, as an implicitly private *Field*, and initialised once for the whole class. Initialisation expressions are most often simple literals, but may be complex invocations, as shown here. However, they cannot rely on the values of any other *Fields* in the same *Class*, which may not yet have bound values. In this example, we assume that *rates* refers to a shared *Field* declared in the superclass *Account*, which is guaranteed to be initialised before *netRate* in the subclass *SavingsAccount*.

A *Field* may be initialised to the result of object creation. The following makes *boss* a shared field, initialised to a particular *Person* instance:

```
<Field name="boss" shared="true" type="Person" visible="private">
  <Create creator="make" implicit="true" type="Person">
    <Literal value="John" type="String"/>
    <Literal value="Monkfish" type="String/>
  </Create>
</Field>
```

Finally, the only kind of *Field* you may declare in an *Interface* is one that is *shared*, *public* and initialised. This is a public constant of the interface. The following is an example of this, possibly declared in *Geometry*, an interface for geometry:

```
<Field name=”pi” shared="true" type=”Decimal” visible=”public”>
  <Literal value="3.14159265358979323846" type="Decimal"/>
</Field>
```

4.3 Method Declaration

The *Method* element may be used to define a named method of a *Class* or *Interface*. A *Method* is a kind of *Member* (from which it obtains a *visibility*) and transitively a kind of *Property* (from which it obtains a name and a type). A *Method* may declare one or more formal arguments and a result type, representing a signature. A *Method* may also declare a body expression to compute its result. A *Method* may also declare generic type parameters, if it manipulates generic arguments or results. An *Interface* may only contain method signatures, also called *abstract* methods.

The grammar for the *Method* element is:

```
<!ELEMENT Method (Generic*, Variable*, Assert*,
    Sequence, Rescue?)>
<!ATTLIST Method name NMTOKEN #REQUIRED>
<!ATTLIST Method abstract (false | true) "false">
<!ATTLIST Method override (false | true) "false">
<!ATTLIST Method visible (private | protected | public) "public">
<!ATTLIST Method type NMTOKEN #REQUIRED>
```

The attributes of the *Method* element are:

- *name* – the name of the method (required);
- *abstract* – whether the method is abstract (optional);
- *override* – whether a redefined method (optional);
- *visible* – the visibility of the method (optional);
- *type* – the result type of the method (required).

The *name* must be in *camelCase*, the expected style for all property names. If the *abstract* attribute is *true*, this indicates that the method is a signature; it may not have a *Sequence* body. If the *override* attribute is *true*, this indicates that the method is a redefinition of an existing concrete method; but it is illegal to override a *private Method*. The *visible* attribute may be set to any of: *private* (visible in the declaring class), *protected* (also visible in descendant classes) or *public* (visible to all). By default, a *Method* will be *public*. The *type* records the result type of the method and may be any legal ReMoDeL OOP type name (see section 2).

The *Method* element has the children:

- *Generic* – additional generic type parameters (optional, zero-to-many);
- *Variable* – the formal arguments of the method (optional, zero-to-many);
- *Assert* – the preconditions and postconditions (optional, zero-to-many);
- *Sequence* – the method's single body expression (optional, when concrete);
- *Rescue* – the method's recovery statements (optional).

A *Method* need only declare additional generic parameters, over and above the *Generic* parameters already declared in its owning *Class*, if the objects it manipulates are of some other generic type. A *Method* may declare zero to many formal arguments, since the owning object is implicitly the first argument. A *Method* may declare zero to many *Assert* statements, representing its preconditions and postconditions. If the *Method* is *abstract* (a signature), it may not declare any body *Sequence*, but if it is concrete (implemented), it must declare a single *Sequence* expression, representing the method body. The type of the *Sequence* must match the type of the *Method*, that is, if the method is procedural, the type will be *Void* and if the method is functional, some other result type will be declared. The *Method* may optionally declare a *Rescue* statement, containing recovery code.

Simple examples of the *Method* element include:

```
<Method name="getAge" type="Natural" visible="public">
  <Sequence type="Natural">
    <Return type="Natural"/>
      <Identifier name="age" scope="object" type="Natural"/>
    </Return>
  </Sequence>
</Method>

<Method name="setAge" type="Void" visible="public">
  <Variable name="age" type="Natural"/>
  <Sequence type="Void">
    <Assign symbol="equals"/>
      <Identifier name="age" scope="object" type="Natural"/>
      <Identifier name="age" type="Natural"/>
    </Assign>
  </Sequence>
</Method>

<Method name="birthday" type="Void" visible="public">
  <Sequence type="Void">
    <Assign symbol="plus"/>
      <Identifier name="age" scope="object" type="Natural"/>
    </Assign>
  </Sequence>
</Method>
```

These examples illustrate getter and setter methods. The first example illustrates how to return a result (see section 3.12). The second and third examples illustrate binary assignment and the unary in-place incrementing assignment (see section 3.4). Note in the second example how the same *Identifier* name may be reused to refer both to an object field and to the related formal argument that was used to supply its value. This is possible because *Identifier* distinguishes the scope of the variables that it references.

The following shows how to represent abstract method signatures, as they might appear in an *abstract Class*, or *Interface* declaration.

```
<Method name="setAge" abstract="true" type="Void"
    visible="public">
  <Variable name="age" type="Natural"/>
</Method>
```

```
<Method name="getAge" abstract="true" type="Natural"
    visible="public"/>
```

For access methods with no arguments, no child contents are needed; for methods with arguments, one or more *Variable* children should be declared. It is also legal to add *Assert* statements to signatures.

The following method *deposit* also has *Assert* statements expressing its pre- and postconditions. These ensure that the *amount* deposited is positive, and that the *balance* was updated as a result of calling the method:

```
<Method name="deposit" type="Void" visible="public">
  <Variable name="amount" type="Integer"/>
  <Assert contract="positive amount" when="before" type="Void">
    <Operator symbol="moreThan" type="Boolean">
      <Identifier name="amount" type="Integer"/>
      <Literal value="0" type="Integer"/>
    </Operator>
  </Assert>
  <Assert contract="balance increased" when="after" type="Void">
    <Operator symbol="equals" type="Boolean">
      <Identifier name="balance" state="after" scope="object"
            type="Integer"/>
      <Operator symbol="plus" type="Integer">
        <Identifier name="balance" state="before" scope="object"
            type="Integer"/>
        <Identifier name="amount" type="Integer"/>
      </Operator>
    </Operator>
  </Assert>
  <Sequence type="Void">
    <Assign symbol="plus" type="Void">
      <Identifier name="balance" scope="object" type="Integer"/>
      <Identifier name="amount" type="Integer"/>
    </Assign>
  </Sequence>
  <Rescue type="Void">
    <Assign symbol="equals" type="Void">
      <Identifier name="balance" state="after" scope="object"
            type="Integer"/>
      <Identifier name="balance" state="before" scope="object"
            type="Integer"/>
    </Assign>
  </Rescue>
</Method>
```

This example also demonstrates, through the use of a *Rescue* block, how to roll back the state of the object, which we assume is an instance of *CurrentAccount*, to its prior state, in case the *deposit* action fails. This *Rescue* does not specify any *restart* attempts, but will simply pass on the failure to the caller.

The body of a *Method* may refer to all identifiers that are in scope. This includes all object *Fields* that are visible to the method, all of the method's formal arguments, and any local *Variables* that were introduced. If a *Method* needs to refer to the whole of the owning object, this is via the special *Identifier* called *self*. Referring to *self* is useful when passing the current object as an argument to a foreign method.

In the context of inheritance, a *Method* may override an earlier *Method* having the same name. In this case, it is possible for the method body of the overriding method to refer to the earlier version, through the special *Identifier* called *super*. This is known as super-method invocation, and may occur at any convenient point in the overriding Method:

```
<Method name="openWith" type="Void" override="true"
      visible="public">
  <Variable name="amount" type="Integer"/>
  <Sequence type="Void">
    <Invoke method="openWith" type="Void">
      <Identifier name="super" scope="special" type="Account"/>
    </Invoke>
    <!-- rest of method body omitted for brevity -->
  </Sequence>
</Method>
```

We assume that this illustrates the *openWith* method of a *SavingsAccount*, which overrides the same-named method of *Account*, but uses that class's method to perform basic account opening actions, before proceeding to do further actions of its own (not illustrated above).

4.4 Creator Declaration

The *Creator* element may be used to define a named initialising procedure for a *Class*. A *Creator* is a kind of *Member* (from which it obtains a *visibility*) and transitively a kind of *Property* (from which it obtains a name and a type). A *Creator* may declare one or more formal arguments and always declares a body expression to initialise an instance of its owning class. The result type of a *Creator* is always *Void*, since it is evaluated for the side effect of object initialisation. A *Creator* may also declare generic type parameters, if it accepts generic arguments. A *Creator* is always concrete, never abstract. An *Interface* may not define a *Creator*.

The grammar for the *Creator* element is:

```
<!ELEMENT Creator (Generic*, Variable*, Assert*, Sequence)>
<!ATTLIST Creator name NMTOKEN #REQUIRED>
<!ATTLIST Creator override (false | true) "false">
<!ATTLIST Creator visible (protected | public) "public">
<!ATTLIST Creator type NMTOKEN #FIXED "Void">
```

The attributes of the *Creator* element are:

- *name* – the name of the creator (required);
- *override* – whether a redefined creator (optional);
- *visible* – the visibility of the creator (optional);
- *type* – the result type of the creator (required).

In ReMoDeL OOP, a *Creator* is a named procedure. The *name* must be in *camelCase*, the expected style for all property names. If the *override* attribute is *true*, this indicates that the *Creator* is a redefinition of an existing one (with the same

name). The *visible* attribute may be set to any of: *protected* (visible in the defining class and descendant classes) or *public* (visible to all), but not *private*, which makes no sense in a *Creator*. By default, a *Creator* will be *public*. The *type* of a Creator is always *Void*.

The *Creator* element has the children:

- *Generic* – additional generic type parameters (optional, zero-to-many);
- *Variable* – the formal arguments of the creator (optional, zero-to-many);
- *Assert* – the preconditions and postconditions (optional, zero-to-many);
- *Sequence* – the creator's single body expression (optional, when concrete).

A *Creator* need only declare additional generic parameters, over and above the *Generic* parameters already declared in its owning *Class*, if its arguments are of some other generic type. A *Creator* may declare zero to many formal arguments, since the target object is implicitly the first argument. A *Creator* may declare zero to many *Assert* statements, representing its preconditions and postconditions; but it makes no sense to *Rescue* a *Creator*, since no valid object has yet been created that could be restored. A *Creator* must declare a single *Sequence* expression, representing the initialising body. The initialising expressions are all *Assign* statements.

Simple examples of the *Creator* element include the *default* creator, which has no arguments:

```
<Creator name="create" type="Void" visible="public">
  <Sequence type="Void"/>
</Creator>
```

This will perform no initialisation beyond whatever default values were declared for the owning *Class's* fields. The default creator is usually called *create* by convention. It need not be redefined for each class, unless it is desired to override the default initialisation rules (see section 4.2). Another example is the *standard* creator, which initialises an object's fields from a set of arguments:

```
<Creator name="make" type="Void" visible="public">
  <Variable name="temperature" type="Integer"/>
  <Variable name="scale" type="String"/>
  <Sequence type="Void">
    <Assign symbol="equals" type="Void">
      <Identifier name="temperature"
          scope="object" type="Integer"/>
      <Identifier name="temperature" type="Integer"/>
    </Assign>
    <Assign symbol="equals" type="Void">
      <Identifier name="scale" scope="object" type="String"/>
      <Identifier name="scale" type="String"/>
    </Assign>
  </Sequence>
</Creator>
```

This is the most common kind of *Creator*. Note how simple *Assign* statements are used to perform the initialisation. By convention, the *standard* creator is always called *make*. It is common for *make* to be redefined in each subsequent subclass, as further initialisation arguments are added. The following illustrates the *Creator* for a *Student*, which extends the *Creator* for a *Person* (see section 2.4):

```
<Creator name="make" override="true" type="Void" visible="public">
  <Variable name="forename" type="String"/>
  <Variable name="surname" type="String"/>
  <Variable name="gender" type="Character"/>
  <Variable name="age" type="Natural"/>
  <Variable name="registration" type="Natural"/>
  <Create creator="make" type="Person">
    <Identifier name="super" scope="special" type="Person"/>
    <Identifier name="forename" type="String"/>
    <Identifier name="surname" type="String"/>
    <Identifier name="gender" type="Character"/>
    <Identifier name="age" type="Natural"/>
  </Create>
  <Assign symbol="equals" type="Void">
    <Identifier name="registration" scope="object"
        type="Natural"/>
    <Identifier name="registration" type="Natural"/>
  </Assign>
</Creator>
```

It delegates to the superclass creator to perform most of the initialisation (of *Fields* that were declared in the superclass), and then it initialises the last *Field* that was declared in the subclass *Student*. Where a *Creator* invokes the inherited *Creator*, the super-creation instruction must always appear first in the *Creator* body. This is because many translations insist on this convention.

4.5 Package Declaration

ReMoDeL OOP uses the ReMoDeL Package and Namespace Model for organising class and interface definitions into separate packages. See the separate model specification [3] for full details.

A ReMoDeL OOP package is declared using the *Package* element to wrap the contents of the package. A skeleton example is the following:

```
<Package model="OOP" name="People" location="example.people">

  <!-- other definitions inserted here -->

</Package>
```

This declares that the definitions contained within belong to the package whose namespace is *People*, whose contents are to be stored at the logical location *example.people* and that the package model type is *OOP*. This information may be interpreted differently by different translators: they may use the namespace identifier to qualify all definitions, or use the logical location to store all compiled definitions at a given physical location under a root directory for the target language.

In OOP, a package expresses its dependency upon other packages using the *Consult* dependency, after which each class or interface refers to other types using explicit *Employ, Inherit* and *Satisfy* references. This is because most translations need to express reference dependencies between classes and interfaces at a finer-grained level than whole package scope:

```
<Consult model="OOP" package="People" location="example.people"/>
<Consult model="OOP" package="Banks" location="finance.banks"/>
<Consult model="OOP" package="Equity" location="finance.equity"/>

<Class name="SavingsAccount" visible="public">
  <Inherit refer="Account" kind="Class" from="Banks"/>
  <Satisfy refer="Asset" kind="Interface" from="Equity"/>
  <Employ refer="Person" kind="Class" from="People"/>
  <!-- rest of class definition omitted for brevity -->
</Class>
```

The only package that is conventionally imported into the current namespace using the global *Import* dependency instruction is the *Core* package:

```
<Import model="OOP" package="Core" location="lib.core"/>
```

This contains the basic types: *Byte, Boolean, Character, Integer, Short, Long, Natural, Decimal, Void* and the classes *Object, String* and the root *Interface*. Importing this package allows class and interface types to employ all the basic types without declaring explicit *Employ* dependencies. In translations, most of these types are built-in types, but some translations will need to import the *String* datatype for each class in the current package.

While it would be possible to *import* other packages globally, this would lead to translations with very long import clauses at the head of each class file. In any case, even though this would obviate the need to declare explicit *Employ* dependencies upon the used types, classes still need to express their *Inherit* and *Satisfy* dependencies upon other classes or interfaces explicitly.

4.6 Program Declaration

In ReMoDeL OOP, the notion of a program is deliberately flexible. A program consists of a set of class and interface declarations, possibly originating from various packages, of which one class will serve as the driver of the system. In principle, we adhere to Meyer's tenet that "real systems have no top" [4], that is, we should expect the main entry point of an object-oriented system to change from time to time, indeed, for there to be possibly many different entry points into the system.

For this reason, any class having a default creator called *create* and a method of no arguments called *execute* may be chosen as the entry point. In this way, more than one class may serve as the entry point on different occasions, so long as the tools know which class to choose, when generating the system. If there are many candidate driver classes, the name of the chosen driver must be supplied as a parameter to the generation tools. If there is only one candidate driver, the tools should be able to detect this automatically.

Generators will produce a short main program that first constructs a default instance of the driver class using the creator *create*, and then invokes the *execute* method to set

the whole system running. The *create* creator is responsible for creating and initialising the other objects that constitute the system. The *execute* method initiates processing among the objects of the system. Whereas Eiffel has a similar approach, in which a root creation procedure creates and also initiates the system [4], we prefer to distinguish *constructing* the system from *executing* it, treating these as separate steps, since in certain languages (such as C++), it is problematic to think of executing a system that has not yet finished being constructed.

An example skeleton of a *Driver* class is illustrated below (the name of the class is not significant), showing how both the creator and the method must be declared public:

```
<Class name="Driver" visible="public">
  <Creator name="create" type="Void" visible="public">
    <!-- creator body omitted for brevity -->
  </Creator>
  <Method name="execute" type="Void" visible="public">
    <!-- method body omitted for brevity -->
  </Method>
  <!-- other fields and methods omitted -->
</Class>
```

How the result of the computation is communicated to the end-user is a matter for different translations and may depend on the supplied core libraries. The OOP model is expected to be supplied with a number of standard libraries, including an I/O library with the package name *InOut*. This is expected to define standard input and output stream classes, which behave in a similar way across different translations. Every effort will be made to ensure a common semantics for OOP translations.

5 References

[1] World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0, 5th edition*, eds. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, 26 November, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/

[2] A. J. H. Simons, *ReMoDeL Functional Programming Model Specification*, Technical Report, Department of Computer Science, University of Sheffield, 2011.

[3] A. J. H. Simons, *ReMoDeL Package and Namespace Model Specification*, Technical Report, Department of Computer Science, University of Sheffield, 2011.

[4] B. Meyer, *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall, 1997.