The University Of Sheffield.

ReMoDeL

Package and Namespace

Model Specification

Version: 0.5

Date: 02 November 2011

Anthony J H Simons
Department of Computer Science
University of Sheffield

1 Introduction

This document describes the specification for ReMoDeL PKG, a dialect of XML [1] used to encode the Package and Namespace Model. This is a foundational model, used as a core subset by other ReMoDeL XML dialects.

1.1 Model Scope

The Package and Namespace Model is intended to support a common language for organising models and software artefacts in packages. Different ReMoDeL XML dialects may support quite different kinds of model, ranging from high-level conceptual models to low-level implementation models, but all dialects expect some kind of over-arching organisation of model artefacts into packages, to manage the partitioning of artefacts across different logical and physical spaces and to provide a common scheme for cross-referencing these artefacts from other packages.

ReMoDeL PKG was developed in order to satisfy common requirements for package and namespace management across all ReMoDeL XML dialects. The language has grown in response to the need to reuse whole models and model artefacts at different levels of granularity. By providing a single ReMoDeL PKG dialect, the aim is to foster a common approach to managing packages and namespaces, and remove this concern from the specification of other ReMoDeL dialects, which may instead refer to this specification.

1.2 Model Semantics

ReMoDeL PKG handles issues of identification across different kinds of model, and within each kind of model. At the outermost layer, it must distinguish between artefacts that belong to different kinds of model (such as OOP, DBQ, FUN). Within any given model, the main areas of concern are to find flexible ways of labelling model artefacts that belong to different namespaces; and to find ways of mapping logical namespaces to physical storage locations on a computer's backing store.

Since each ReMoDeL dialect has a unique 3-letter identifier, it was easy to satisfy the first requirement by labelling packages according to the *primary* kind of model they represent. Packages that include artefacts from *subsidiary* model-kinds that are explicit subsets of the primary model-kind (for example, PKG is a subset of many other model-kinds) may do so; and packages then use the identifier for the primary model-kind.

Namespaces were more of a challenge, since we desired to support translations into languages that use both flat namespaces and hierarchical namespaces. In the end, we chose a mapped model, in which flat namespace identifiers could be mapped to logical location identifiers, expressing the abstract hierarchical structure of the namespace. The location information may also be used to create physical directory structures for storing translated models and generated code.

Referring to artefacts in different namespaces uses a dependency language to express the logical provenance of the referenced artefact. The language distinguishes the wholesale importing of foreign packages from the reuse of individual artefacts from

foreign packages. Imported elements are typically unique in the using context, but may also be distinguished through their namespace identifiers.

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 PKG 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 Packages and Dependency

ReMoDeL uses a single kind of *Package* that adapts to every XML dialect used in the project. A package is just one kind of *Classifier*, an abstract concept that defines a namespace. *Classifiers* depend on various kinds of *Declaration* (such as *Type* or *Property* declarations) according to different flavours of *Dependency*. The ancestry of these elements is illustrated in the fragment of the ReMoDeL Metamodel shown in figure 1.

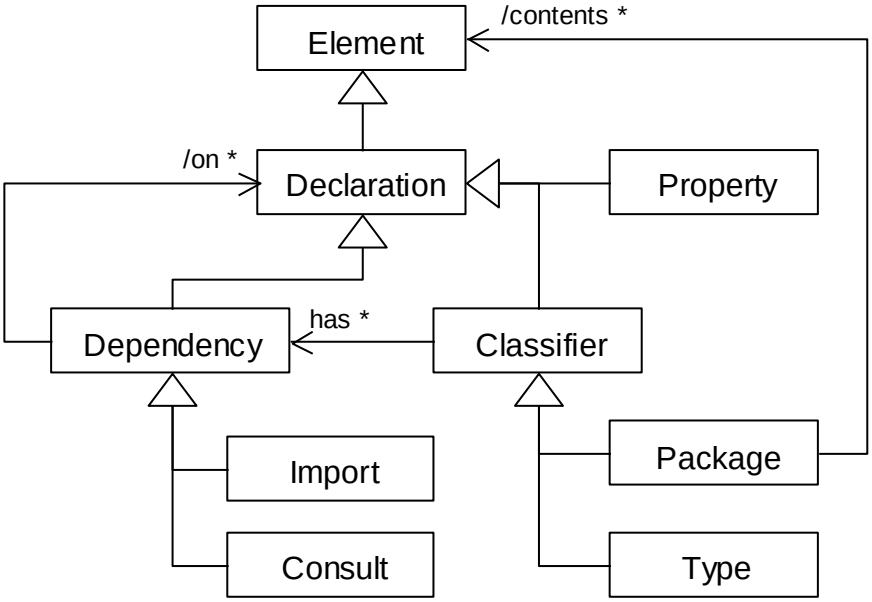

Figure 1: Package Dependency in the ReMoDeL Metamodel

All ReMoDeL packages are defined as instances of the *Package* metaclass. A package encapsulates elements known as the package's contents, different kinds of *Element* that vary according to the model-kind. By virtue of being a kind of *Classifier*, a package defines a unique namespace (within its model-kind), which is used as a namespace for the package's contents. The intention is that, within each model-kind, package names refer to unique packages; and within each package, element names refer to unique elements.

Relationships between packages (and other kinds of element) are expressed as instances of children of the abstract *Dependency* metaclass. *Import* and *Consult* describe coarse-grained package-level dependencies between one *Package* and another. Further fine-grained dependencies also exist (see section 3).

2.1 Package Declaration

The *Package* element is used to declare a package, the modular translation unit used in ReMoDeL to encapsulate the contents of a model. Packages are used to impose model-kinds and logical namespaces upon ReMoDeL models. They may also be translated into equivalent packages or modules in a target language.

The *Package* element is the root of every model. XML tools should expect to validate a root *Package* element in the XML file storing the package. Packages are complete once defined, in the sense that there is no other location where further contents may be found for the same package. This does not prevent tools from building a package incrementally, but does mean that the clients of a package must be able to assume that it is the whole package. Packages are usually named, but one package (the default package) may be unnamed; its location is also empty, and is assumed to be the current working directory.

The grammar for the *Package* element is:

```
<!ELEMENT Package (ANY*)>
<!ATTLIST Package model NMTOKEN #REQUIRED>
<!ATTLIST Package name NMTOKEN #IMPLIED>
<!ATTLIST Package location CDATA #IMPLIED>
<!ATTLIST Package library (false | true) false>
```

The attributes of the *Package* element are:

- *model* – the model-kind of the package (required);
- *name* – the namespace defined by the package (optional);
- *location* – the abstract location for the package (optional);
- *library – true*, if this package is a standard library (optional).

The *model* must be given as a model-kind used to identify models in the ReMoDeL project. Examples include: OOP, FUN, DBQ, PKG, always in *UPPERCASE*. The *name*, when supplied, must be given in *CapitalCase* (see section 1.4) and defines a namespace that must be unique for the model-kind. Namespaces are the abbreviated short names by which ReMoDeL packages are recognised. The *location*, when supplied, must be given as an abstract pathname consisting of *lowercase* identifiers separated by periods. Locations are hierarchical identifiers by which ReMoDeL packages are recognised. The *library* flag is *false* by default, but set to *true* to indicate a library package (a model that describes predefined library code).

The children of the *Package* element are multiple, and vary from model to model:

- any kind of model element (optional, zero-to-many).

For example a FUN package might define a collection of *Basic*, *Variable* and *Function* elements, whereas an OOP package might define a collection of *Class*, *Interface*, *Basic* and *Symbolic* elements. There is no significance to the order of a package's contents, although the definition-before-usage style is preferred. Library packages only declare the signatures of the elements they contain, since they serve to describe predefined code, and no new code is generated from them.

Examples of the *Package* element include:

```
<Package model="OOP" name="Core" location="lib.core"
    library="true">
  <!-- contents omitted -->
</Package>

<Package model="OOP" name="Util" location="lib.util"
    library="true">
  <!-- contents omitted -->
</Package>
```

These are library packages in the *OOP* model called *Core* and *Util*. They are stored at abstract locations *lib.core* and *lib.util*, which may translate to directory trees with a shared root. Further examples include:

```
<Package model="FUN" name="List" location="lib.util.list"
    library="true">
  <!-- contents omitted -->
</Package>

<Package model="FUN" name="Main" location="my.work.proj.main">
  <!-- contents omitted -->
</Package>
```

These are packages for the *FUN* model. The abstract location *lib.util.list* must be completely disjoint with similarly named *OOP* locations *lib.core, lib.util*. The last example shows a user-defined package. Finally, one unnamed package may exist:

```
<Package model="OOP">
  <!-- contents omitted -->
</Package>
```

This is the default package in the OOP model. There can only be one default package for each model. This is stored in the root directory (for the model) and may map to generated code in the root directory (for the target language).

2.2 Import Dependency

The *Import* element is used to import the contents of a foreign package. This is a coarse-grained dependency that relates the current (home) package to another (foreign) package. The instruction is interpreted as opening the foreign package and including its contents in the namespace of the home package, as if they had been defined there. It is most useful in ReMoDeL dialects where all dependencies are handled at package-level (such as FUN). In other ReMoDeL dialects (such as OOP), detailed dependencies between types are also recorded.

The grammar for the *Import* element is:

```
<!ELEMENT Import EMPTY>
<!ATTLIST Import model NMTOKEN #IMPLIED>
<!ATTLIST Import package NMTOKEN #REQUIRED>
<!ATTLIST Import location CDATA #REQUIRED>
```

The attributes of the *Import* element are:

- *model* – the model-kind of the package (optional);

- *package* – the name of the package to be imported (required);
- *location* – the abstract location for the named package (required).

The *model*, when supplied, must be given as one of a finite number of model-kind identifiers used in the ReMoDeL project, in *UPPERCASE*. By default, the foreign model-kind is assumed to be the same as the home model-kind. The value of *package* is a package name in *CapitalCase* (see section 1.4). The value of *location* is an abstract pathname consisting of *lowercase* identifiers separated by periods.

The *Import* element has no children. As a matter of style, package imports are listed at the head of the home package's contents.

Simple examples of the *Import* element include:

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

<Import model="FUN" package="List" location="lib.util.list"/>
```

These import the named packages into the namespace of the current unspecified package. These examples use the explicit model-kind attribute. Further examples of the *Import* element used in a package context include:

```
<Package model="FUN" name="Main" location="my.work.proj.main">
  <Import package="List" location="lib.util.list"/>
  <!-- other contents omitted -->
</Package>

<Package model="OOP" name="Sorting" location="example.sorting">
  <Import package="Util" location="lib.util"/>
  <!-- other contents omitted -->
</Package>
```

These examples import the named foreign packages into the home context described by the containing *Package* elements. In both cases, the model-kind of the imported packages is left implicit, and is inferred to be the same as the home package.

2.3 Consult Dependency

The *Consult* element is used to open a foreign package, without importing its contents into the current namespace. This is a coarse-grained dependency that relates packages to other packages. It is a more subtle instruction than the *Import* declaration, an advance notification that the home package depends on elements from the foreign package. *Consult* may be used with other dependency declarations that request individual declarations from the consulted package.

The grammar for the *Consult* element is:

```
<!ELEMENT Consult EMPTY>
<!ATTLIST Consult model NMTOKEN #IMPLIED>
<!ATTLIST Consult package NMTOKEN #REQUIRED>
<!ATTLIST Consult location CDATA #REQUIRED>
```

The attributes of the *Consult* element are:

- *model* – the model-kind of the package (optional);
- *package* – the name of the package to be consulted (required);
- *location* – the abstract location for the named package (required).

The *model*, when supplied, must be given as one of a finite number of model-kind identifiers used in the ReMoDeL project, in *UPPERCASE*. By default, the foreign model-kind is assumed to be the same as the home model-kind. The value of *package* is a package name in *CapitalCase* (see section 1.4). The value of *location* is an abstract pathname consisting of *lowercase* identifiers separated by periods.

The *Consult* element has no children. As a matter of style, package consultations are listed at the head of the home package's contents.

Simple examples of the *Consult* element include:

```
<Consult model="OOP" package="Util" location="lib.util"/>

<Consult model="FUN" package="List" location="lib.util.list"/>
```

These consult the named packages from the namespace of the current unspecified package. These examples use the explicit model-kind attribute.

A further example of the *Consult* element used in a package context is:

```
<Package model="FUN" name="Main" location="my.work.proj.main">
  <Consult package="List" location="lib.util.list"/>
  <Consult package="Func" location="higher.func"/>

  <!-- other contents omitted -->

  <Employ refer="List" kind="Record" from="List"/>
  <Employ refer="head" kind="Function" from="List"/>
  <Employ refer="tail" kind="Function" from="List"/>
  <Employ refer="map" kind="Function" from="Func"/>
  <Employ refer="filter" kind="Function" from="Func"/>

  <!-- other contents omitted -->

</Package>
```

This example consults two packages named *List* and *Func*, opening the abstract locations where these packages are to be found. At some later point, various *Employ* dependencies refer to the exact elements to be used. This allows translations to selectively import the declarations from the foreign packages, rather than importing all declarations wholesale. The *Employ* element is defined below (see section 3.1) and describes a usage-dependency on some element from the named package.

3 Classifiers and Reference

In certain ReMoDeL dialects, it is necessary to express dependency at a finer scale than the coarse-grained package-level dependency described above (see section 2). This is particularly true in languages that generate code in modular units that are smaller than a package, such as ReMoDeL OOP, which is modularised around *Class* and *Interface* datatypes. These translation units are invariably kinds of *Classifier*, so the notion of dependency is defined at this level, for all classifiers.

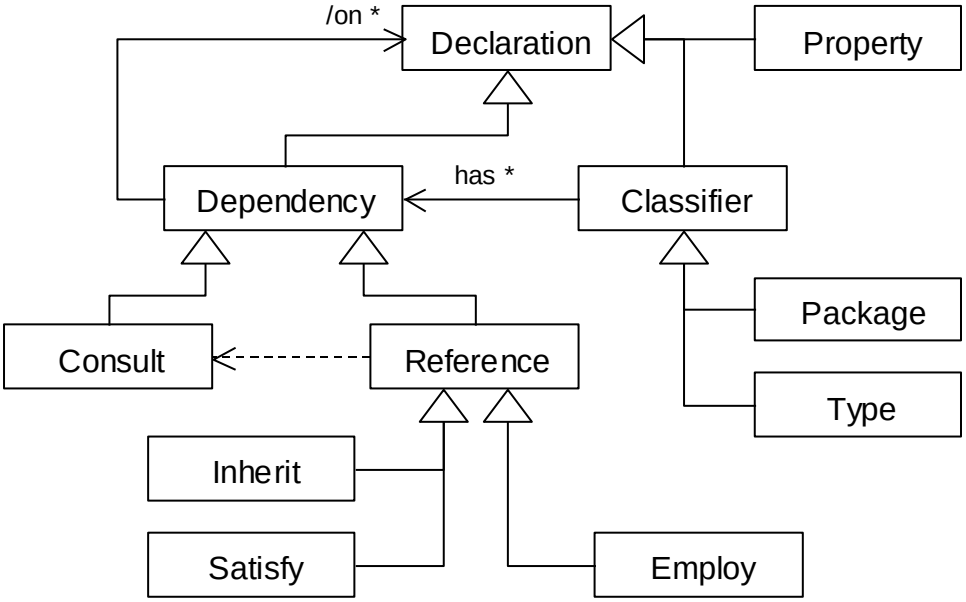

Figure 2: Reference Dependency in the ReMoDeL Metamodel

The finer-grained dependencies described here are all kinds of cross-reference. These are modelled as children of the abstract *Reference* dependency and include the *Employ*, *Inherit* and *Satisfy* dependencies. All of these presume that the appropriate packages have been consulted, such that the cross-referenced names may be mapped to their full definitions in the foreign packages.

3.1 Employ Reference

The *Employ* element is used to express a usage reference dependency upon a declaration. This is a fine-grained dependency that relates a classifier to another declaration. For example, a structured type such as a class may refer to a primitive type or another class as part of its definition, so may need to express this dependency. *Employ* is used in a context where the foreign package has been consulted (see 2.3).

The grammar for the *Employ* element is:

```
<!ELEMENT Employ EMPTY>
<!ATTLIST Employ refer NMTOKEN #REQUIRED>
<!ATTLIST Employ kind NMTOKEN #IMPLIED>
<!ATTLIST Employ from NMTOKEN #REQUIRED>
```

The attributes of the *Employ* element are:

- *refer* – the identifier of the related declaration (required);
- *kind* – the metaclass of the related declaration (optional);
- *from* – the package owning the related declaration (required).

The *refer* identifier is the employed element's usual name, either in *camelCase* (if a property) or in *CapitalCase* (if a classifier – see section 1.4). The value of *kind* is the metaclass of the employed element. The value of *from* is the name of the package owning the element.

The *Employ* element is a reference, and has no children.

Examples of the *Employ* element include:

```
<Employ refer="List" from="Lists"/>

<Employ refer="head" kind="Function" from="Lists"/>

<Employ refer="Set" kind="Class" from="Util"/>

<Employ refer="HashSet" from="Util"/>
```

These examples illustrate how to represent a usage dependency upon some other kind of element that is known to exist in the named package. The package identifier can only be understood in the context of a previous consultation. The name allows the exact element to be identified. The metaclass specifier is used to speed up access to the named element (by filtering the metaclass kind).

Different ReMoDeL dialects may use the *Employ* element in different ways, referring to declarations both outside and within the same package. This is determined by how much explicit dependency information the model requires.

3.2 Inherit Reference

The *Inherit* element is used to express an inheritance reference dependency upon another classifier. This is a fine-grained dependency that relates a classifier to another classifier. For example, a class may inherit from a parent class, whose inherited fields and methods are added to the child class. Other kinds of classifier could support an inheritance relationship. *Inherit* is used in a context where the foreign package has been consulted (see 2.3).

The grammar for the *Inherit* element is:

```
<!ELEMENT Inherit EMPTY>
<!ATTLIST Inherit refer CDATA #REQUIRED>
<!ATTLIST Inherit kind NMTOKEN #IMPLIED>
<!ATTLIST Inherit from NMTOKEN #REQUIRED>

The attributes of the Inherit element are:
```

- *refer* – the identifier of the related declaration (required);
- *kind* – the metaclass of the related declaration (optional);

- *from* – the package owning the related declaration (required).

The *refer* identifier is the inherited classifier's usual name in *CapitalCase* (see section 1.4). The value of *kind* is the metaclass of the inherited classifier. The value of *from* is the name of the package owning the classifier.

The *Inherit* element is a reference, and has no children.

Examples of the *Inherit* element include:

```
<Inherit refer="Person" kind="Class" from="People"/>

<Inherit refer="Account" from="Finance"/>
```

These examples each represent an inheritance dependency upon another classifier that is known to exist in the named package. The package identifier can be understood only in the context of a previous consultation. The metaclass specifier may speed up searching for the referenced classifier.

3.3 Satisfy Reference

The *Satisfy* element is used to express an interface-satisfaction reference dependency upon another classifier. This is a fine-grained dependency that relates a classifier to another classifier. For example, a class may satisfy an interface, whose signatures it supports; and an interface may extend another interface, adding to its signatures, thereby also satisfying the other interface. *Satisfy* is used in a context where the foreign package has been consulted (see 2.3).

The grammar for the *Satisfy* element is:

```
<!ELEMENT Satisfy EMPTY>
<!ATTLIST Satisfy refer CDATA #REQUIRED>
<!ATTLIST Satisfy kind NMTOKEN #IMPLIED>
<!ATTLIST Satisfy from NMTOKEN #REQUIRED>

The attributes of the Satisfy element are:
```

- *refer* – the identifier of the related declaration (required);
- *kind* – the metaclass of the related declaration (optional);
- *from* – the package owning the related declaration (required).

The *refer* identifier is the satisfied classifier's usual name in *CapitalCase* (see section 1.4). The value of *kind* is the metaclass of the satisfied classifier. The value of *from* is the name of the package owning the classifier.

The *Satisfy* element is a reference, and has no children.

Examples of the *Satisfy* element include:

```
<Satisfy refer="List" from="Util"/>

<Satisfy refer="Asset" kind="Interface" from="Finance"/>
```

These examples each represent an interface satisfaction reference dependency upon another classifier that is known to exist in a given package. The package identifier can only be understood in the context of a previous consultation.

4 Logical to Physical Mapping

The logical structure for packages described in previous sections is intended to support an intuitive mapping to physical directories and files. The various XML files representing ReMoDeL models need to be stored in predictable locations. Likewise, the code generated from models in different target languages will need to be stored according to a similar scheme. The expectation is that target language compilers will be able to observe the same modular structure as described for logical ReMoDeL packages. In this way, generated modules will be able to refer to each other and also to predefined library modules, which will be found in predictable locations.

4.1 Mapping Package Models to Files

Packages are expected to be stored in XML files, whose names are derived from the logical names of the packages, and placed in a directory, whose structure is determined from the type of the model. For example, the package model:

```
<Package model="OOP" name="Core" location="lib.core"
    library="true">
  <!-- contents omitted -->
</Package>
```

is expected to be stored in a file named *Core.xml* that is placed inside a directory with the hierarchical structure *xml/oop*, so giving the complete path: *xml/oop/Core.xml* for the package model. Similarly, the package model:

```
<Package model="FUN" name="List" location="lib.util.list">
  <!-- contents omitted -->
</Package>
```

is expected to be found in the directory *xml/fun* and have the file name *Lists.xml*, so giving the complete path: *xml/fun/Lists.xml* for the package model. Directory paths consist of identifiers in *lowercase* separated by the platform's file separator character. Package file names are capitalised by convention. Some operating systems may choose to disregard case.

The reason for this particular directory structure is because the tools processing models and possibly generating code will generally wish to access both model files and files in proprietary programming languages (such as Java, C++ or SQL), which will be placed in a similar directory structure. As a result, the tools will most likely be launched in a working directory above this structure.

Tools will read and write files placed in different directories according to the type of data. The file-type for model files is initially *xml*, and then models are distinguished further by the kind of model, here reflected in the intermediate directories *oop* or *fun*. This allows for the eventual possibility that packages in different models may be given the same simple package names. Package names must be guaranteed to be unique within a given model, but may be reused across models.

4.2 Mapping Generated Code to Files

Generated code is expected to be stored in files, whose names are derived from the name of the compilation unit in the target language, and placed in a directory, whose structure is determined from the target language and the hierarchical structure of the package. Complete paths may vary in style and length according to the preference of the target language. For example, the following package model:

```
<Package model="FUN" name="Func" location="higher.func">
  <!-- contents omitted -->
</Package>
```

may contain logical models of higher-order functions, with names such as *map, filter* and *reduce*. The translation of these into Haskell assumes that the whole package will become a single module, with the logical Haskell module name: *Higher.Func*, which is usually mapped to a directory structure: *Higher/Func.hs* by Haskell compilers. The complete path is however: *haskell/Higher/Func.hs*, since this must also include the target language name as the prefix of the path. This is used to distinguish software generated for Haskell from software generated for Lisp or ML, which may use the prefix names *lisp* and *ml* respectively.

Languages that use whole packages as their translation units will typically adopt this approach. Languages that use smaller translation units will adopt a slightly different approach. For example the following package model:

```
<Package model="OOP" name="Finance" location="example.finance">
  <!-- contents omitted -->
</Package>
```

may contain logical models of classes and interfaces, with names such as *Asset*, *Account* and *CurrentAccount*. The translation of these into Java assumes that each will be placed in a separate file, with names such as: *Asset.java*, *Account.java* and *CurrentAccount.java*. To reflect the logical package structure, each of these will also be declared as belonging to the Java package: *example.finance* and so must be placed in a similar sub-directory structure. The target language name is added as the prefix of the directory path, yielding the complete paths:

```
java/example/finance/Asset.java
java/example/finance/Account.java
java/example/finance/CurrentAccount.java
```

In general, the case conventions must be appropriate to the target language. The C# language generally prefers capitalised names for package related pathnames. The only fixed convention is that the target language type is always given in *lowercase* as the prefix to any path. For code generated in C++ or C#, conventional prefix names may be used, such as *cplus* and *csharp*, since symbols are illegal in pathnames.

4.3 Mapping Library Packages to Files

As part of any ReMoDeL installation, standard libraries will be provided for each final target language. Standard libraries are useful, in that they provide the boilerplate code that is used in every generated system, and so reduce the workload for code generators. They also provide the middleware for linking language-independent

interfaces to the native standard libraries in each target language. For example, the ReMoDeL OOP *Core* package defines a standard *String* model with a standard set of methods. This must be mapped onto a library *String* class in each target language, which serves as a wrapper or adapter around the native *String* provided by each target language, which may otherwise provide a slightly different interface in every target language.

Library package models must be accessible by the ReMoDeL tools, which may need to check expressions against published interfaces in library models. For this reason, library packages are described using the same package syntax, but need only contain signatures for the elements they contain. For example, the library package model:

```
<Package model="OOP" name="Core" location="lib.core"
    library="true">
  <!-- contents omitted -->
</Package>
```

is frequently referenced by other OOP package models, and may be found at the standard location: *xml/OOP/Core.xml* as stated above. It defines a set of foundation classes and interfaces that belong to the logical package *lib.core*, including classes such as *Object, String, SystemError* and *BrokenContract*.

Likewise, target language compilers must be able to find the predefined library code corresponding to these models. The target code for these classes and interfaces is therefore placed in predictable locations, similar to the scheme described above for generated code (see section 4.2). For example, in the *Java* translation, the following paths must lead to predefined library classes:

```
java/lib/core/Object.java
java/lib/core/String.java
java/lib/core/SystemError.java
java/lib/core/BrokenContract.java
```

The conventions for library files should be similarly predictable for translations into other languages that use package-sized translation units.

4.4 Mapping Dependency Declarations

Dependency declarations will be translated according to the conventions of the target language. This could happen in a variety of ways:

- the language may not have an internal language for expressing dependencies, but may require an external configuration file, e.g. Eiffel;
- the language may refer to packages by standard names and from these will construct absolute pathnames leading to foreign code, e.g. Java or Haskell;
- the language may express every dependency explicitly and generators may construct relative pathnames leading to all referenced code, e.g. C++.

Where possible, the distinction between *importing* and *consulting* a package should be preserved. This will have different effects in different target languages. For example, in Haskell, the translation of a PKG *Import* declaration:

```
<Import model="FUN" package="Func" location="higher.func"/>
```

is the following:

```
import Higher.Func
```

This includes all the contents of the imported module in the current namespace. To get the effect of PKG *Consult* with limited *Employ* directives:

```
<Consult model="FUN" package="Func" location="higher.func"/>
<Employ refer="map" kind="Function" from="Func"/>
<Employ refer="filter" kind="Function" from="Func"/>
```

you have to generate the following Haskell translation, which only includes the named functions in the current namespace:

```
import Higher.Func (map, filter)
```

In an object-oriented language like Java, the translation of a PKG *Import* declaration affects more than one generated class file. For example, importing the whole *Core* library package:

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

must always be translated as:

```
import lib.core.*;
```

at the head of every generated Java file in the home package. To get the effect of PKG *Consult* with limited *Employ* directives:

```
<!-- Dependency expressed at package level -->
<Consult model="OOP" package="Core" location="lib.core"/>

<!-- Dependency expressed later in some class -->
<Employ refer="Object" kind="Class" from="Core"/>
<Employ refer="String" kind="Class" from="Core"/>
```

you have to fold these declarations into selective import statements at the head of every dependent class or interface:

```
import lib.core.Object;
import lib.core.String;
```

It may also be possible to support kinds of dependency that do not import the named declarations into the home package's namespace (see below).

4.5 Mapping Namespace Declarations

A namespace is simply a way of distinguishing two code entities whose simple names might otherwise clash. Even a simple flat namespace model allows every code entity to be referenced by a qualified binary naming scheme: *prefix:name* if ever the need to disambiguate a simple *name* arises. ReMoDeL packages offer the possibility of generating a flat namespace from the package *name*, or a hierarchical namespace from the package *location*. Different target languages may support flat namespaces,

hierarchical namespaces or even no namespaces at all. The following illustrates how to handle the translation of namespaces in each of these situations.

For example, in a Java translation, the ReMoDeL package *location* is used as a hierarchical namespace for the generated class. Given the package declaration:

```
<Package model="OOP" name="Finance" location="example.finance">
  <!-- contents omitted -->
</Package>
```

every generated class in that package should then be placed in this Java namespace, using a package declaration at the head of each class file, whose hierarchical structure mimics the directory location of the file:

```
package example.finance;
```

Client Java code in any other namespace than this will typically have to import declarations from this namespace, before it may use these classes. Some other target languages in this family include Haskell and C#.

By contrast, the usual idiom in C++ is to use a flat namespace, which is unrelated to the location of the code. For this, the ReMoDeL package *name* is used to define a C++ namespace:

```
namespace Finance {
    // Other C++ definitions placed in here
}
```

Client C++ code in any other namespace than this will need *using* declarations to identify the classes from this namespace:

```
using Finance::Account;
using Finance::SavingsAccount;
```

Note that this can be completely independent of the file dependency issue. In C++, whereas every file dependency (on home or foreign declarations) must be recorded explicitly (see above), *using* declarations need only be generated for foreign declarations that live in a different namespace.

Finally, a few target languages do not yet support namespaces, in which case translations have to simulate namespaces using special naming conventions. One example is the Eiffel language, which supports packaged software clusters, yet these all exist in a single default namespace. This proved to be an issue when porting Eiffel to the .NET environment. The chosen solution was to lengthen all identifiers, qualifying all names with an additional prefix, corresponding to the flat namespace name. In Eiffel, the above classes would simply be defined as:

```
class FINANCE_ACCOUNT ...

class FINANCE_SAVINGS_ACCOUNT...
```

Thereafter, they would be used as usual in the target language. The only issue for translators in this context is to maintain a map from simple to qualified names.

4.6 Mapping Qualified Package Access

While the usual approach to translation expects target languages to import, or selectively import elements from foreign packages into the home package, ReMoDeL also supports the possibility of referring to elements by their fully qualified package names. This can be for two reasons:

- because it is desired not to import the element into the current namespace, which would seemingly pollute the namespace;
- because it is desired to refer unambiguously to elements that have clashing simple names, but come from different namespaces.

In this case, the foreign package must always be *consulted* in ReMoDeL. This has the effect of making the declarations in that package available, should the current translation need to refer to elements within it. For example, the *Consult* declaration:

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

has the effect of making the elements in this library package available. The normal approach thereafter is to use *Employ* directives; but if this is not done, the types and variables in the package may still be referred to using fully qualified names in XML. For this, the package name is used as a simple flat namespace prefix, which maps to the full hierarchical namespace, in a style appropriate to each target language:

- *Core:Object* maps to *lib.core.Object*
- *Core:String* maps to *lib.core.String*
- *Finance:Account* maps to *example.finance.Account*
- *Finance:Asset* maps to *example.finance.Asset*
- *Func:map* maps to *Higher.Func.map*
- *Func:filter* maps to *Higher.Func.filter*
- *List:head* maps to *Lib.Util.List.head*
- *List:tail* maps to *Lib.Util.List.tail*

The notion is that simply prefixed names would be translated to the equivalent fully qualified names in the target language, wherever they are encountered. This kind of qualification may be used wherever a simple property name or classifier name was otherwise expected, for example:

```
<Variable name="account" type="Finance:CurrentAccount"/>
```

This shows how to disambiguate which *CurrentAccount* class is denoted in a context where the *Finance* package has been consulted, along with another package that also defines another *CurrentAccount*. In Java, this would be translated into:

```
example.finance.CurrentAccount account;
```

In other words, the fully qualified name would be used in all cases. In this way, it is possible to refer unambiguously to the *CurrentAccount* type from either package without confusion.

The same approach generalises to other constructions using qualified variable names or qualified function names. For example, the following assumes that the FUN package *lib.util.list* has been consulted, but names have not been imported:

```
<Apply function="List:head" type="Integer">
  <Identifier name="list" type="List:List[Integer]"/>
</Apply>
```

In Haskell, this would be translated into a special qualified import statement:

```
import qualified Lib.Util.List as List
```

and then the code translation would use *List.head, List.List* as the locally qualified names for the *head* function and *List* record type. The *Variable, Identifier* and *Apply* elements used in these examples are documented elsewhere [2].

Of course, processing namespaces in the body of a ReMoDeL model puts a greater burden on the model transformation tools, which need to be sensitive to XML prefixes wherever they encounter any kind of type name, variable name or function name. This may require more sophisticated multiple passes through the model, if special qualified forms of import statement are required, as in Haskell.

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.