
This version is available at: http://eprints.mdx.ac.uk/6220/

Copyright:

Middlesex University Research Repository makes the University's research available electronically.

Copyright and moral rights to this work are retained by the author and/or other copyright owners unless otherwise stated. The work is supplied on the understanding that any use for commercial gain is strictly forbidden. A copy may be downloaded for personal, non-commercial, research or study without prior permission and without charge.

Works, including theses and research projects, may not be reproduced in any format or medium, or extensive quotations taken from them, or their content changed in any way, without first obtaining permission in writing from the copyright holder(s). They may not be sold or exploited commercially in any format or medium without the prior written permission of the copyright holder(s).

Full bibliographic details must be given when referring to, or quoting from full items including the author’s name, the title of the work, publication details where relevant (place, publisher, date), pagination, and for theses or dissertations the awarding institution, the degree type awarded, and the date of the award.

If you believe that any material held in the repository infringes copyright law, please contact the Repository Team at Middlesex University via the following email address:

eprints@mdx.ac.uk

The item will be removed from the repository while any claim is being investigated.

See also repository copyright: re-use policy: http://eprints.mdx.ac.uk/policies.html#copy
A pattern based approach to defining translations between languages

Girish Maskeri1, James Willans1, Tony Clark2, Andy Evans1, Stuart Kent3, Paul Sammut1
Contact: girishmr@cs.york.ac.uk

Abstract. The 2U Consortium have recently submitted a proposal for the definition of the UML 2.0 infrastructure. This uses an innovative technique of rapidly “stamping out” the definition using a small number of patterns commonly found in software architecture. The contribution of this paper is to introduce the idea of reusability of mappings between languages and defining some of the reusable mapping templates. This paper also illustrates how these templates can be used to stamp out mapping between languages by stamping out a mapping between UML and Java.

1. Introduction

Crucial to successful software development is the ability to translate between different representations. The most prolific illustration of this is the compilation of high-level languages into assembler. With the increasing trend of modelling designs of software prior to its implementation using languages such as UML, there is also a need to provide translations between software models and their implementation. A desirable characteristic of these modelling languages is that they abstract from a particular implementation, but can be translated into any implementation. Consequently, it is necessary to provide (and prove) many translations and provide new translations to mirror the development of new languages. The current approach to creating these translations (often referred to as mappings) is to bespoke each one in view of the two languages being translated. This is a time consuming and error-prone process. In this paper we propose that an alternative approach is to construct models of different languages by using patterns, and to automatically derive translations between languages by using predefined translations between these patterns.

This work forms part of a larger project that is defining a submission for UML 2.0 [1]. An issue with UML is not only the translation (and an assurance of the integrity of the translation) between models and an implementation, but also different models (profiles) as it is widely expected that UML2.0 will be a family of languages [2]. The submission for UML 2.0 [3] draws heavily on the philosophy that well formed languages exhibit recurring structures, and reuses a small set of patterns (encapsulated into templates) to “stamp out” the whole of the UML 2.0 infrastructure. The pattern based approach to defining the UML infrastructure can also be applied to models of other languages. This paper exploits the fact that different modelling languages (collected together in profiles) are produced using common patterns (a library of templates). We define the mapping between the elements of the

1 Department of Computer Science, University of York, UK
2 Department of Computer Science, Kings College, London, UK
3 Department of Computing, University of Kent, Canterbury, UK
template library and use these mappings to automatically derive the mapping between the profiles.

2. Overview

The principle idea behind the approach presented in this paper is that models of languages can be built from a set of patterns describing concepts, and translations between the concepts. Consequently, the predefined mappings enable translations to take place between languages with ease. For instance, a common pattern is the description of the inheritance mechanism. For some languages this will be described as a singular inheritance pattern, for others multiple inheritance pattern. It is common to want to translate between languages that support multiple inheritance and languages that support singular inheritance, therefore a mapping is provided between the two realisations of the inheritance mechanism. When a language uses multiple inheritance (C++), it can be translated to a language that uses singular inheritance (Java) by invoking this mapping. This is illustrated in figure 1.

![Figure 1. Overview of translation approach](image)

3. Templates approach to language definition

The work described in this paper is an extension of the 2U consortiums submission to defining the infrastructure for UML 2.0 [3]. In this section we briefly describe the approach in order to provide a context for the contribution of the paper.

One of the distinguishing characteristics of the 2U consortiums approach [4] is to clearly separating the syntax and the semantics of languages. The syntax of models and their semantics are described as distinct entities related by a mapping (semantic mapping). In the infrastructure definition, the syntax is described as abstract, the abstract syntax will be mapped to a concrete syntax within the super structure (i.e. boxes and lines).

It is generally considered that good software architectures exhibit recurring structural patterns [5].
The 2U approach identifies this and uses an innovative technique of “stamping out” the infrastructure using patterns commonly found in software architecture. These patterns are encapsulated into reusable libraries called package templates that are described using precise class diagrams [3] and the OCL (object constraint language). The use of class diagrams and OCL enables a high degree of perspicuity compared to traditional formal representations of semantics (i.e. denotational semantics using Lamda calculus). Therefore, using a small library of templates [3], complex languages (including the UML 2.0 infrastructure) can be rapidly “stamped out” and understood with minimal effort. Moreover, since templates are a rich form of reuse, “stamped out” language are known to be correct in view of the correctness of the templates. An example of a common pattern is shown in figure 2. This describes how one element (the container) conceptually contains another (the contained). The philosophy behind this approach has been proved by way of a powerful meta-modelling tool (MMT) [6].

This paper illustrates how the template-based approach can be extended for defining and reusing mapping between languages. The mappings between patterns (mapping templates) are encapsulated into packages (mapping package). These mapping can then be “stamped out” along with the languages being defined.

4. Relations concept for defining mapping

The essence of a translation is a function from one representation to another. Illustrated in figure 3 is a template which captures this function.
**5. Defining Mappings**

**5.1 Approach**

In this section, we extend the generic template presented previously to derive a template that maps between structures commonly found in modelling and programming languages. Two important structures are the container pattern (illustrated in figure 2) and the generalisation pattern. Here we define a mapping between these patterns which will be used in section 5.2 to stamp out the mapping between fragments of two languages: UML and Java. The process of defining a mapping between the container pattern of modelling languages and container pattern in programming languages using the generic mapping template (figure 3) is illustrated in figure 4.

The Contains template/pattern on the left side shown in figure 3 can be used to derive containment structures in modelling languages like UML (classes contain attributes, and so on) and the template on the right can be used to derive corresponding structures in programming languages like Java. The ContainerMap template is a mapping between the containers and extends the generic mapping template as shown in figure 4. In addition to the classes and association, well formedness rules expressed as OCL...
constraints are used to express properties and constraints on the mappings. The complete definition is presented in Appendix A in textual format.

A template that maps a multiple inheritance pattern to a delegation pattern is as shown in figure 5.

![Diagram of mapping generalisation patterns in modelling and programming languages](image)

**Figure 5.** mapping generalisation patterns in modelling and programming languages

The left hand side of figure 5 shows the most common generalisation pattern in modelling languages and on the right hand side is one of the generalisation patterns found in programming languages and is the delegation model [7,8]. Element3 delegates to element2, this is achieved through the uses and implements relation. The DelegationInheritanceMap template maps these two patterns. Again, in addition to the classes and association, well formedness rules expressed as OCL constraints are used to express properties and constraints on the mappings. The complete definition is presented in Appendix A in textual format.

Other Patterns in languages have been identified and defined as templates by the 2U consortium [4].

### 5.2 Example

By combining the ContainerMap template and the DelegationInheritanceMap template defined in the previous section, we can derive a mapping package for mapping between multiply inherited containers in one language to delegated, inherited containers in another. This mapping is entirely reusable and can be applied between any pairs of languages where this property holds. Of course, there are many other ways in which multiple inheritance in one language can be translated into single inheritance structures [9]. However, this pattern identifies a commonly used approach – subclass delegation.

Applying this template to UML and Java results in the general mapping shown in figure 6.
In UML both sublassing as well as subtyping is achieved through inheritance. In Java one of the schemes is to use delegation for subclassing and interfaces for subtyping.

The substitutions required to “stamp out” the UML-Java mapping package using the ContainerMap and DelegationInheritanceMap templates is illustrated in figure 7. Here, UMLClass is substituted for the Container to be mapped, whilst UMLAttribute is its Contained element. JavaClass and JavaAttribute are substituted into the same structure in the target of the mapping. The stamped out mappings package as viewed in the MMT [5] (the meta modelling tool) is shown in figure 7. Note that for brevity, all the stamped constraints and translation methods have been omitted.
6. Discussion

This paper has described some of the first attempts by the 2U group to address the challenge of reusing mappings between languages. We began by motivating the need for reusing mappings between languages, as they are essential to so many aspects of software engineering. The approach we used to achieve this goal has been to define a generic relations template (as a model-level concept) and to utilise this to describe mappings between common structures in languages. We have claimed that we can derive the mapping between two languages from the pre-defined mappings between the patterns of the two languages. We have illustrated this by deriving a simple mapping between UML and Java from the generic mapping templates. This mapping implements a common approach to relating multiple inheritance and single inheritance containers.

Clearly, there is much work to be done to show the scalability of the approach. For example, whilst the relations approach is clearly advantageous in terms of its mathematical properties, stamping out relations at the model level can result in quite verbose models (see Figure 7). We are therefore investigating the definition of relations as a first class UML modelling construct. Such a construct would share many of the
properties of an association, but would also permit properties such as the nesting of relations to be expressed as well.

Finally, we believe that the ability to reuse patterns of mappings is an essential first step towards realising the OMG’s vision for MDA (Model-Driven Architecture) [10]. MDA is fundamentally based on the ability to rapidly generate mappings between platform independent and platform dependent modelling languages. Currently, the bespoke approach used by vendors to define mappings is simply too time consuming and error prone to realise this goal.

Acknowledgments
This work has been generously funded by the TCS (Tata Consultancy Services), India.

References

[8] J. Viega, P. Reynolds, B Tutt, R Behrends, Multiple inheritance in class Based Languages.
Appendix A: Mapping Templates

A.1. Pair template

Package Pair(A,B)
  class <<A>> end

  class <<B>> end

  class <<A + "x" + B>>
  <<A>> : Pair::<<A>>;
  <<B>> : Pair::<<B>>;
end
end

A.2. Relation Template

package Relation(R,A,B) extends (Relations::Pair)(A,B)
  class <<R>>
    pairs : Set(Relation::<<A + "x" + B>>) ;
    dom : Set(Relation::<<A>>) ;
    ran : Set(Relation::<<B>>) ;

    "lookup" + A>(b:Relation::<<B>>):Set(Relation::<<A>>)
    self.pairs->select(p | p->at(1) = b)->collect(p | p->at(0))
  end

    "lookup" + B>(a:Relation::<<A>>):Set(Relation::<<B>>)
    if self.pairs = Set{}
      then Set{}
      else
        self.pairs->select(p | p->at(0) = a)->collect(p | p->at(1))
      endif
    end

  image():Set(Relation::<<B>>)
  self.pairs->collect(p | p->at(1))
end

  inverseImage():Set(Relation::<<A>>)
  self.pairs->collect(p | p->at(0))
end

  isFunctional():DataTypes.Boolean
  self.pairs->forall(p | self.pairs->forall(q | q->at(0) = p->at(0)
    implies p=q))
end

  isInverseFunctional():DataTypes.Boolean
  self.pairs->forall(p | self.pairs->forall(q | q->at(1) = p-
    at(1) implies p=q))
end

  isInjection():DataTypes.Boolean
  self.isFunctional() and self.isInverseFunctional()
end

  isOnto():DataTypes.Boolean
  if self.image() = self.ran
    then true
  else
    false
  endif
end
isTotal(): DataTypes.Boolean
  self.inverseImage() = self.dom
end

isBijection(): DataTypes.Boolean
  self.isInjection() and self.isOnto()
end

A.3. Map Template

package Map(R,A,B) extends (Relations::Relation)(R,A,B)
class <<R>>
  "to" + B>(a:Map::<<A>>):Map::<<B>>
  let cached = self."lookup" + B>(a)
in
    if not cached->isEmpty
    then cached.selectElement()
    else
      let b = Map::<<B>>.new(Seq{})
in
        self.pairs := (self.pairs->including(Seq{a,b})) []
b
    end
  endif
end
end
end

A.4. ContainerMap Template

package ContainerMap (containerRelation, Container1, Contained1, ContainedRelation, Container2, Contained2)
extends
  (Templates::Contains)(Container1,Contained1),
  (Templates::Contains)(Container2,Contained2),
  (Relations::Map)(containerRelation,Container1,Container2),
  (Relations::Map)(ContainedRelation,Contained1,Contained2)
class <<containerRelation>>
  <<ContainedRelation>> : ContainerMap::<<ContainedRelation>> ;
  init(s:Seq(Instance)):Object
    self.<<ContainedRelation>>:= (ContainerMap::<<ContainedRelation>>.
      new( Seq {})) [] self end

  "to"+Container2>(a:ContainerMap::<<Container1>>):ContainerMap::
  <<Container2>>
  let b = super.run(a)
  Contained2s = a."+" + "s">->collect(b |
    self."to" + ContainedRelation>."to" + Container2>(b))
in b."+" + "s">:= Contained2s [] b end
end
end
A.5. DelegationInheritanceMap Template

package DelegationInheritanceMap ( GElem1, generalisableRel, GElem2, subclassRel, subClassingElem, subTypeRel, subTypingElem, GElem2subTypingElemRel )
extends
  (Relations::Map)(generalisableRel,GElem1,GElem2),
  (Relations::Map)(subclassRel,GElem1,subClassingElem),
  (Relations::Map)(subTypeRel,GElem1,subTypingElem),
  (TemplateLibrary::Generalisable)(GElem1),
  (TemplateLibrary::Generalisable)(subTypingElem),
  (TemplateLibrary::RelatedManyToMany)(GElem2,GElem2subTypingElemRel,subClassingElem)

////Each class in UML maps to a class in Java. It displays a bijective relation
class generalisableRel
  inv isBijective
    self.isBijective() = true
    fail: "java model not proper"
  end
end

//// The elements of the domain are UMLclasses
////elements of range are Java attributes.
////According to the figure 3, a UML class is associated with a java attribute because of the delegate relation. The class is associated with an attribute only if the umlclass has at least one parent. There are always root classes in any model hence it is partial function.
class subclassRel
  inv isPartialInjective
    self.isInjective() = true and self.isTotal = false
    fail: "java model not proper"
  end
end

"to" + GElem2>>(a:DelegationInheritanceMap::<<GElem1>>)
if (a.parents)
  then
    super.run(a)
  endif
end

////To capture the subtyping information every umlclass gives rise to a javaInterface. The java interface extends the interface derived from the parent of the UMLclass as shown in figure 3. It is bijective.

class subTypeRel
  inv isInjective
    self.isInjective() = true
    fail: "java model not proper"
  end
end
