Int. Computer Symposium, Dec. 15-17, 2004, Taipei, Taiwan.

Strategies for Translating UML/OCL Design Models to JAVA/JML Designs

Ali Hamie
School of Computing, Mathematical and Information Sciences,
University of Brighton, Brighton, UK.
{a.a.hamie@brighton.ac.uk}

Abstract. *The Object Constraint Language OCL is a textual notation that can be used for making UML models more precise by expressing formal constraints on the modelling elements that occur in UML diagrams. OCL can be used to specify invariants on classes and preconditions and postconditions of operations and methods. The Java Modeling Language JML is a behavioural interface specification language for specifying Java classes and interfaces. Like OCL, JML can be used to specify invariants and preconditions and postconditions. However JML explicitly targets Java, whereas OCL is not specific to any one programming language.*

This paper deals with the translation of some aspects of UML design models with OCL constraints to Java classes and interfaces annotated with JML assertions. Rather than giving a particular translation, the paper proposes different translation strategies that would be possible. A set of defaults for all the decisions would be adopted which would allow translation to be automated, for example by a tool that could take the UML/OCL model and translate it directly into an initial JML/Java design that could later be modified as desired.

Keywords: OCL, UML, JML, Constraints

1. Introduction

The Unified Modeling Language UML [2] is widely accepted as the standard for object-oriented modelling and is supported by a number of CASE tools. The Object Constraint Language OCL [8] is a part of UML, and was introduced to formally express additional constraints on object-oriented models that diagrams cannot convey by themselves. OCL can be used to specify invariants on classes and preconditions and postconditions of operations.

The Java Modeling Language JML [5][6] is a formal specification language specifically developed for specifying and describing the detailed design and implementation of Java modules (classes and interfaces) [1]. It is model-based supporting, in particular, class invariants, and method specification by precondition and postcondition to document required module behaviour. There are various tools that support the checking and manipulation of JML specifications including a run-time assertion checker. A description of the different tools available can be found in [3].

Following the lead of Eiffel [7], the assertion language of JML is based on side-effect free Java expressions. The language is extended with few operators and constructs including operators for universal and existential quantifications that are essential for making the language more expressive. JML also provides a library of mathematical models (sets, bags, sequences, etc.) defined as pure Java classes that are intended to be used in specifications.

This paper presents and discusses different strategies for translating UML/OCL design models into JML/Java designs consisting of Java classes and interfaces annotated with JML assertions. The paper also adopts a set of defaults for the translation. This makes it possible for the translation to be automated by a tool that could take the UML model with OCL constraints and translate it directly into an initial JML/Java design that could later be modified as desired. The main focus will be on translating the modelling elements of class diagrams with associations that are directed. The translation facilitates reasoning about the specification and the verification and testing of the Java classes using a wide range of tools that manipulate JML. And because JML/Java preserves most features of the object-oriented structure of UML/OCL models, errors detected within the JML/Java specification produced by the translation could more easily be traced back into the initial UML/OCL model.

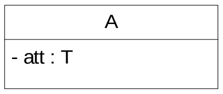
The rest of the paper is organised as follows. Section 2 presents various choices for translating classes, attributes, and invariants. Section 3 shows how associations with various multiplicities are translated. Section 4 shows how aggregation and composition are translated. Section 5 shows how association classes are translated. Section 6 shows how generalisation is translated. Section 7 shows how operation specifications are translated. Section 8 provides the conclusion.

2. Translating UML classes

In this section we discuss various options for translating UML class diagrams into JML. A class in a class diagram is mapped to a Java class of the same name, annotated with JML assertions.

2.1 Class attributes

The attributes of a class are mapped to fields of the same name and with the appropriate types. Attribute type declarations are required for translation to JML/Java. Figure 1 shows a class named A with attribute att of type T, where T is assumed to be a basic value type, i.e. one of the following types: Boolean, Integer, Real, or String.



```
public class A {
 private /*@ spec_public @*/ T' att;}
```

Figure 1. A class and its translation to JML

The class A is mapped to a Java class with the same name A and the attribute att is mapped to a field of the same name att of type T', where T' is the translation of type T. JML specifications are included in the code as annotations which are comments starting with //@ or starting with /*@ and ending with @*/. For example, the annotation /*@ spec_public @*/ indicates that for the purpose of specification the field att is public. The type of the corresponding Java field depends on the type T. If T is Boolean, the corresponding field type is boolean. If T is Integer, the corresponding field type is one of the following types: byte, short, int, or long. Given that Integer is the type of mathematical integers, it is appropriate to translate it as long, by default. If T is Real, the corresponding field type is one of the following types: float or double. We also translate Real as double, by default. If T is String, the corresponding field type is String. And because the type String is an object (reference) type, an additional constraint is needed on the field att that says att cannot take the value null since it is declared to be total. This constraint can be expressed using either the annotation /*@ non_null @*/ on the declaration as follows:

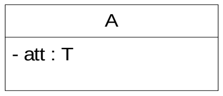
```
public class A {
  private /*@ spec_public non_null @*/
                          String att; }
```

or as an explicit invariant as follows:

```
public class A {
  private /*@ spec_public @*/ String att;
  //@ public invariant att != null;}
```

JML invariants are boolean expressions and follow the keyword invariant. The modifier public indicates that the invariant is public. Using annotations on the declaration makes the specification simpler and easier to read. For this reason, this translation will be the default.

The disadvantage of using private fields as public for specification purposes is that the specification is coupled with these fields so that any changes to them will affect the specification. This can be overcome by using model fields which are used only in the specification. This is shown in Figure 2 where a represents clause is added which indicates how the value of the model field att is obtained from the concrete field att_c.



```
public class A {
  //@ public model T' att;
      private T' att_c;
  //@ private represents att<- att_c;}
```

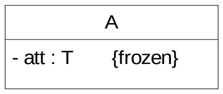
Figure 2. An alternative translation for class A

If the type T is an enumerated type with values v1 and v2, the corresponding type is simply the Java type T declared as enum T {v1, v2}.

If the attribute att is optional, i.e. declared as att : T[0..1], the corresponding type is the wrapper class for the basic types. For example, if T is Integer, the corresponding type would be one of the following types: Byte, Short, Integer, or Long. By default, Integer is translated as Long. If T is Real, the corresponding type would be one of the following: Float or Double. Again by default, Real is translated as Double. If T is String, the field type would be String in which case there is no additional constraint on the field att since it can already have the value null.

Using wrapper classes for the translation may make the corresponding JML assertions more complicated because one needs to unwrap the objects in order to get the actual value. However, this is no longer a problem since the new version of Java makes the unwrapping process automatic.

If the attribute att is marked with UML's {frozen} property as shown in Figure 3, the corresponding field att is declared with the modifier final.



```
public class A {
 private final /*@ spec_public @*/ T' att;}
```

Figure 3. A frozen attribute and its translation

An alternative way is to translate the `frozen` property of the attribute using a history constraint as:

```
//@ constraint att == att@pre;
```

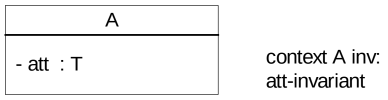
That is, once the object is initialised the value of `att` will be the same in every state.

2.2 Class invariants

Class invariants involving attributes are translated to JML invariants constraining the corresponding fields. Invariants in OCL are expressed as follows:

```
context A
inv: att-invariant
```

where the keyword context indicates the context of the invariant which is the class A in this case, and inv indicates the type of the constraint which is an invariant. The invariant itself att-invariant is a boolean expression which may involve the attribute att. This invariant translates to a JML invariant on the corresponding field `att` as shown in Figure 4, where `att-invariant` is the corresponding JML boolean expression. If a model field is used in the translation, the invariant constrains it.



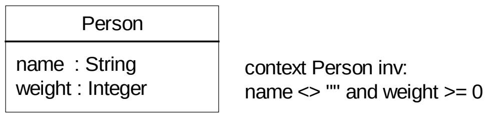
```
public class A {
   private /*@ spec_public @*/ T' att;
   //@ public invariant att-invariant; }
```

Figure 4. A class invariant and its translation

OCL assertions are boolean expressions built using the boolean operators and, or, not, and implies. Quantified expressions are built using the quantifiers forAll and exists. The expressions p and q, p or q, not p, and p implies q are translated to the corresponding JML expressions `p'&&q'`, `p'|| q'`, `!p'`, and `p'==>q'` respectively, where `p'` and `q'` are the translation of the expressions p and q respectively. Quantified expressions are translated using the JML quantifiers `\forall` and `\exists`.

Equality between expressions of the basic types Boolean, Integer, and Real are translated using the Java equality operator `==`. That is `e1 = e2` is translated to `e1' == e2'`, where `e1'` and `e2'` are the translations of e1 and e2 respectively. However, if e1 and e2 are of type String, e1=e2 is translated to `e1'.equals(e2')`. More details about translating OCL expressions and operations into JML can be found in [4].

As an example, Figure 5 shows a class Person with attributes name and weight, and an invariant that says the name cannot be empty and the weight must be greater than or equal to zero. The mapping to JML/Java is also shown in Figure 5, where the name and weight are translated to the fields `name` and `weight` respectively. The inequality name <> "" is translated to `!name.equals("")`.



```
public class Person {
  private /*@ spec_public non_null @*/ String name;
  private /*@ spec_public @*/ int weight;
  //@ invariant !name.equals("") && weight >=0;}
```

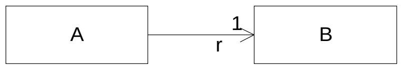
Figure 5. Class **Person** and its translation

3. Translating associations

This section considers associations with various multiplicities and how they are mapped to JML/Java. Associations are used to show how classes are related to each other. Associations are drawn as lines between pairs of classes. The association line may be annotated with role names and multiplicity constraints that indicate how many instances of one class can be linked to an instance of another class. Associations are translated through declared fields in Java classes depending on the navigability specified across the association line. This paper only handles associations that are directed. Bi-directional associations are translated as if they were two separate uni-directional associations with additional constraints.

3.1 Associations with 'one' multiplicity

Figure 6 shows a directed association between classes A and B with role name r at the B's end. This says that an instance of class A is associated with exactly one instance of class B. Translating the association involves translating the classes and the role r. Since the multiplicity of the association is one, r is translated to a field `r` of type `B` that is declared in `A`. An additional constraint is needed to say that `r` cannot be `null`. The mapping is shown in Figure 6 where field `r` is declared as public for specification purposes. This is similar to the attribute case where the attribute type is an object type.



```
public class A {
 private /*@ spec_public non_null@*/ B r;}
```

Figure 6. Association translation

The translation can also be achieved using a model field `r` and a concrete field `rc`, together with a represents clause that defines the model field in terms of the concrete one, as follows:

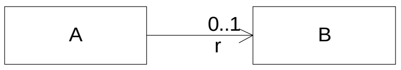
```
public class A {
  //@ public model non_null B r;
  private B rc;
  //@ private represents r <- rc;}
```

The advantage here is that the specification is not tied to the implementation of the class.

The mapping provides a general case for the translation of associations that can be instantiated to specific classes and role names.

3.2 Associations with ‘0..1’ multiplicity

Figure 7 shows an optional (i.e. 0..1 multiplicity) directed association between classes A and B with role name r. This says that an instance of class A might or might not be associated with an instance of class B. In this case the role r is translated to a private field r of type B with no additional constraints because it can already take the value null. The mapping is also shown in Figure 7.



```
public class A {
 private /*@ spec_public @*/ B r;}
```

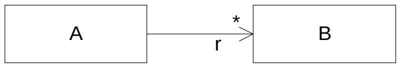
Figure 7. Optional association translation

The translation can also be achieved using a model field r and a concrete field rc, with a represents clause. The translation in this case is similar to that of the association with ‘1’ multiplicity where the non_null constraint is removed.

3.3 Associations with ‘many’ multiplicity

A very common multiplicity in modelling is ‘many’, which is indicated by an asterisk, and means any integer greater than or equal to zero. Figure 8 shows an association between classes A and B with ‘many’ multiplicity and role name r. Each instance of class A is associated or related to a set of instances of class B. JML supports modelling types that can be used in specifications. These include sets, bags and sequences. In order to translate associations with ‘many’ multiplicity, these modelling types are used.

The translation is shown in Figure 8 where the role r is translated to a model field r of type JMLObjectSet. The type JMLObjectSet is the type of finite sets containing objects rather than values. That it treats its elements as object references (addresses) and does not care about the values of these objects. The equality test used by the membership method has uses Java's == operator to compare addresses of these objects. Since JMLObjectSet is defined as a Java class, a constraint is required that restricts the value of the field r to be not null. An additional constraint is also required to say that the elements of r are instances of class B. This is expressed using the universal quantifier \forall, as (\forall Object e; r.has(e);e instanceof B). Note that this translation does not specify how the role r is implemented.



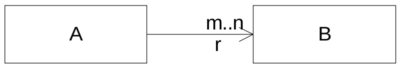
```
public class A {
//@ public model non_null JMLObjectSet r;
/*@ invariant  (\forall Object e;r.has(e);
  @              e instanceof B);
  @*/ }
```

Figure 8. A translation of an association with * multiplicity

Another possible translation is to use the private representations of role r as public for the specification. However, the use of model variables provide abstract and more concise specifications.

If the association is also annotated with {bag} or {seq}, the translation is similar to the set case but uses the types of bags JMLObjectBag and sequences JMLObjectSequence respectively.

Associations with other ‘many’ multiplicities are translated in a similar way, where the multiplicities are reflected in additional constraints constraining the size of such collections. For example Figure 9 shows an association annotated with multiplicity m..n, where m and n are positive integers. In this case the role r is translated to a field r of type JMLObjectSet where the size of r is restricted to be between m and n. The translation is given in Figure 9. The case where the multiplicity is a fixed integer is subsumed within range multiplicity where m and n are equal.



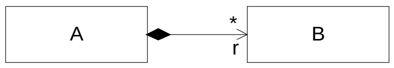
```
public class A {
//@ public model non_null JMLObjectSet r;
/*@ public invariant
  @(\forall Object e;r.has(e);e instanceof B)
  @  && m <= r.size() && r.size() <= n;
  @*/ }
```

Figure 9. A translation of an association with m..n multiplicity

4. Translating aggregation/composition

Aggregations in UML are special associations that represent ‘part-whole’ relationships. The ‘whole’ side of the relationship is often called the aggregate or assembly. Aggregations are specified using a diamond symbol, which is placed next the aggregate. Aggregation is translated as an ordinary association.

Composition is a stronger form of aggregation and implies that instances of the part class may belong to just one instance of the compound class. A composition is shown using a solid (filled-in) diamond, as opposed to an open one. The translation of composition is similar to translating aggregation with an additional constraint to enforce unshared containment. This translation is shown in Figure 10 where the last constraint is used to prevent sharing.



```
public class A {
//@ public model non_null JMLObjectSet r;
/*@ invariant  (\forall Object e;r.has(e);
  @                   e instanceof B);
  @  (\forall A a1, a2;a1 != a2;
  @       a1.r.intersection(a2.r).
  @            equals(new JMLObjectSet());
  @*/ }
```

Figure 10. Composition translation

5. Translating association classes

An association class enables class like features to be added to UML associations. An association class is connected to its association by a dashed line. Such classes may be translated to JML/Java as described above, but with the addition of two fields corresponding to rolenames and types of the classes participating in the association. Depending on the navigability specified across the association line, the participating classes constructs will contain additional fields whose type is a power set of the association class and constrained in size by the multiplicity specified at the opposite association end.

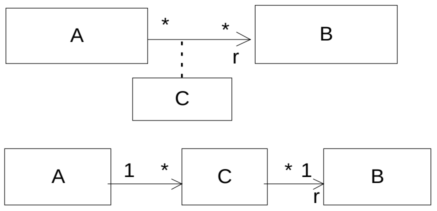
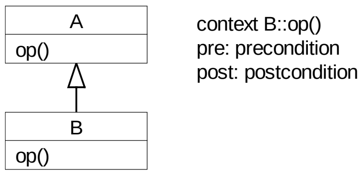


Figure 11. Association class and its transformation to one-many associations

Any pair of classes with a many-to-many association with an association class can be transformed into a model that uses only one-to-many associations as shown in Figure 11. In this case the translation deals only with ordinary associations as covered above. An additional constraint is also needed which says that given an object of type A and another of type B, there is a unique object of C associated with those objects.

6. Translating generalisation

The translation of UML generalisation is straightforward in that Java supports inheritance. Specialised subclass features may then be translated as described earlier. However since Java does not support multiple inheritance, models with such features have to be translated using interfaces. If an operation is redefined in a subclass, its specification in the superclass is inherited. This is indicated in JML by using the keyword `also` as in Figure 12. That is class `B` only shows part of the specification for operation `op`, the other part is specified within class `A`.



```
public class B extends A {
   /*@ also
     @ requires precondition;
     @ ensures postcondition;
  @*/ }
```

Figure 12. Translation of a subclass

7. Translating operation specifications

In OCL operations are specified using preconditions and postconditions which are boolean expressions. The general form of an operation specification is given as follows:

context A::op(p1:T1, ..., pn:Tn)
pre: op-precondition
post: op-postcondition

The first line of the specification defines the class in which the operation is defined, and this is indicated by the keyword context, followed by the signature of the operation. The precondition and postcondition follow the keywords pre and post respectively. This specification is translated to a JML specification of the corresponding method `op` of class `A` as follows:

```
public Class A {
 //@ requires op-precondition;
 //@ ensures op-postcondition;
public void op(T1' p1,...,Tn' pn){...};}
```

The types `T1'`,..., `Tn'` in the signature of the method `op` are the translation of T1, ..., Tn respectively. The precondition op-precondition of method `op` which follows the keyword `requires` is the translation of the OCL precondition op-precondition and the postcondition `op-postcondition` which follows the keyword `ensures` is the translation of op-postcondition.

This will be the default translation of the operation specification. OCL does not support frame conditions that indicate which properties an operation is allowed to modify. However JML supports the description of frame conditions using the assignable or modifies clause. The user can then strengthen the method specification by adding an assignable or modifies clause that indicates which variables the method is allowed to change. In some simple cases this can be deduced by inspecting the postcondition of the operation. However in other cases it is difficult to decide which properties are being modified in order to satisfy the postcondition. In order to make the translation simpler, it is proposed that OCL be extended with a new keyword modifies used in the context of operations to express frame conditions. Query operations that do not alter the state of an object may be indicated by modifies: nothing.

In a postcondition, the expression can refer to values of object properties at the start of the operation or method and upon completion of the operation or method. The value of a property in a postcondition is the value upon the completion of the operation. To refer to the value at the start of the operation, OCL postfixes the property name with the keyword '@pre' as the following example shows:

```
context Person::weightIncreased(n : Integer)
pre: n >= 0
post: weight = weight@pre + n
```

The property weight@pre refers to the value of the property weight of the person object that executes the operation, at the start of the operation.

In JML the operator `\old` is used to refer to the value of an expression at the start of a method. Thus `\old(exp)` denotes the value of the expression `exp` at the start of a method.

In general expressions of the form self.property@pre in postconditions where property is either an attribute or association role are translated to `\old(this.property)`. OCL expressions of the form self.operation@pre(p:T) where operation is a query operation are translated to `\old(this.operation(T' p))`.

The operation oclIsNew is used in postconditions to assert that an object is newly created. That is the expression o.oclIsNew() is true if o is created by the operation and did not exist at precondition time. Such expressions are translated using the JML operator `\fresh` so that exp.oclIsNew() is translated to `\fresh(exp')` where `exp'` is the translation of exp. This indicates that the object denoted by `exp'` is newly allocated.

8. Conclusion

This paper presented and discussed different strategies for translating some aspects of UML/OCL design models to JML/Java designs, that is Java classes and interfaces annotated with JML assertions. The paper dealt with the translation of classes, attributes, invariants, directed associations with various multiplicities, and operation specifications. A set of defaults for the translation has been adopted that would allow it to be automated by a tool that could take the UML/OCL model and translate it directly into a JML/Java design that could later be modified as desired.

One of the benefits of this translation is that it enables the use of JML for the specification of object constraints especially in the detailed design stage of the development of a Java application using UML and OCL. Other benefits include the use of a wide range of tools that support JML for reasoning about specifications, testing and verification of Java programs.

The translation presented in this paper could further be extended to cover the translation of more complex UML constructs such as interfaces, abstract classes, qualified associations and static class features. This should be possible since all these constructs have corresponding representations in JML/Java. Further research needs to be carried out to check whether OCL action constraints can be mapped to JML/Java.

References

[1] K. Arnold, and J. Gosling, The Java Programming Language. The Java Series. Addisson-Wesley, Reading, MA, 2nd edition, 1998.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, The UML User Guide, Addison-Wesley, 1999.

[3] L. Burdy et al., An overview of JML tools and Applications. In Thomas Arts and Wan Fokkink (editors), 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03), pp. 73-89. Volume 80 of Electronic Notes in Theoretical Computer Science <http://www.elsevier.nl/locate/entcs>, Elsevier, June, 2003.

[4] A. Hamie, Translating the Object Constraint Language into the Java Modeling Language. In the proceedings of the 19th ACM Symposium on Applied Computing, 2004.

[5] G. Leavens, et al., JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioural Specifications of Businesses and Systems, Kluwer, 1999.

[6] G. Leavens, A. Baker, and C. Ruby, Preliminary Design of JML: A Behavioral Interface Specification Language for Java. TR #98-06y, revised version June 2004.

[7] B. Meyer, Eiffel: The Language. Object-oriented Series. Prentice Hall, New York, N. Y., 1992.

[8] J. Warmer and A. Kleppe, The Object Constraint Language, second edition, Addison-Wesley, 2003.