TOWARDS VERIFYING JAVA REALIZATIONS OF OCL-CONSTRAINED DESIGN MODELS USING JML

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 formal textual notation that could be used for placing constraints on the modelling elements that occur in UML diagrams. Constraints include invariants on classes and types, and preconditions and postconditions of operations. OCL was designed to be used in conjunctions with UML diagrams resulting in more precise object-oriented designs. The Java Modelling Language (JML) is a behavioural interface specification language designed for specifying Java classes and interfaces. This paper applies OCL for developing Java realizations of UML design models where JML is used as the assertion language. This is achieved by translating a subset of OCL assertions into JML assertions. In order to verify a Java subsystem with respect to a design subsystem with OCL constraints, an appropriate realization relation is defined and the approach is illustrated by an example.

KEY WORDS

Realization Relation, Verification, OCL, JML, Java

1. Introduction

The UML [9] facilitates the precise and formal specification of semantic constraints on object-oriented models using the Object Constraint Language OCL [8][12], and provides notations such as the “realizes” for describing correctness or refinement relationships between different diagrams (or models) at different levels of abstraction. This has improved the possibilities of making program verification more practical. The OCL is a formal textual notation that can be used for constraining modelling elements that occur in UML diagrams. OCL is also used as the constraint language for rigorous software development in the Catalysis approach [4]. The type of constraints that can be expressed using OCL include invariants on classes and types, preconditions and postconditions of operations. The “realizes” relationship asserts that classes (written in a programming language) “realize” the requirements specified in a more abstract class diagram with constraints. This relation allows the programmer to express the correctness of its implementations with respect to UML designs. However there is up to date no possibility of a verification of such relation.

The Java Modelling Language JML [6] is a behavioural interface specification language for specifying Java [1] modules (classes and interfaces). JML assertion language is based on Java expressions and supports various constructs such as universal and existential quantifications that are essential for greater expressiveness. The approach of JML is model based like VDM [5] and Z [11] which means that JML specifications can be expressed abstractly in terms of mathematical models such as sets and relations. However, the mathematical models used in specifications are defined as Java classes making it unnecessary for users to learn another specification language. The advantage of this is that the same notation (in this case Java) which will be familiar to object-oriented programmers is used throughout the specification. JML also supports specification only variables (or fields), which is necessary for complete and expressive specifications. In addition, JML specifications allow frame conditions to be specified [2].

This paper proposes a formalization of the “realizes” relationship with respect to implementations written in Java. This formalization is done in the context of JML. Given a design model represented by a class diagram with OCL constraints, the syntactic and semantic requirements of the relation induced by this model will be defined. The semantical requirements are given by JML specifications which express invariants of classes, and preconditions and postconditions for the implementations of methods. These requirements can then be verified by generating from JML specifications the proof obligations, and proving these obligations using a theorem prover such as PVS [14]. This has become possible by using the tools built by the LOOP project [7][13] which generate proof obligations in PVS from Java code with JML annotations.

Even if in practice such proofs will not be done in full, the approach provides a tool for verifying the critical and important parts of a realization relationship. In addition, the approach is also useful for debugging and testing using the JML tools which include a runtime assertion checker [3] for checking invariants of classes, and preconditions and postconditions of methods.

This paper is organised as follows. Section 2 introduces the design and implementation models of a simple system, and defines a realization relation between such models. It also presents a mapping between OCL and JML assertions and expressions. Section 3 outlines the process of verifying the realization relation. Section 4 provides the conclusion.

2. The General Process

During the process of developing complex software systems various models at different levels of abstraction are produced ranging from analysis models to concrete implementations expressed in terms of some programming language code. This paper focuses on the system design and system implementation and their (formal) relationship.

2.1 The Design Model

Following the Unified Process [10] a design model can be presented by a design subsystem. It is assumed that such subsystem will contain classes with their attributes and operations, associations relations with other classes in such a way that any association is directed and annotated with a role name and multiplicity at the association end, and inheritance. As an essential part of the paper's approach, the subsystem would also include OCL constraints for specifying properties like invariants of classes, preconditions and postconditions for the operations.

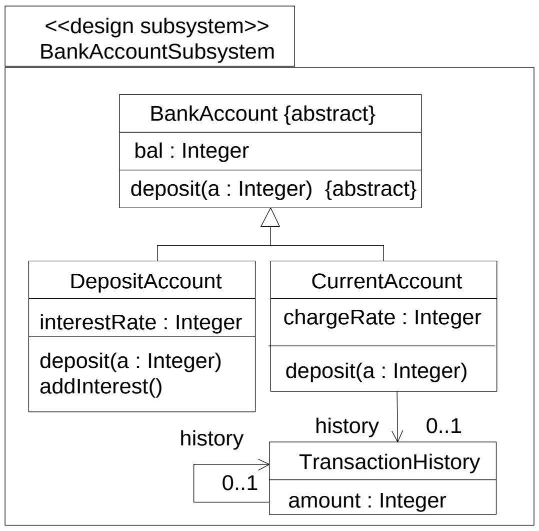

Figure 1. Design model for bank accounts

In order to illustrate the approach, the design model for a (simple) bank account subsystem is considered, and shown in Figure 1. For any current account there is a transaction history that records the amounts of all deposit operations performed on the account. The OCL is used to precisely describe the desired effects of the operations in terms of preconditions and postconditions. In addition, appropriate invariants on the specialised classes will also be expressed using OCL. The OCL constraints are shown in Figure 2.

```
context   BankAccount::deposit(a : Integer)
pre: a >= 0
post: bal = bal@pre + a

context DepositAccount
inv: bal >= 0 and interestRate >=0

context DepositAccount:: addInterest()
post: bal = bal@pre + bal@pre*interestRate* 0.1

context CurrentAccount::
inv: chargeRate >= 0

context CurrentAccount::deposit(a : Integer)
pre: a >= 0
post: history.oclIsNew and
      history.amount = a and
      history.history = history@pre
```

Figure 2: OCL-constraints for account subsystem

```
abstract Class BankAccount
{ protected int bal;
  abstract void deposit(int a);
}

class DespositAccount extends BankAccount
{   private int interestRate;

    public void deposit(int a)
     {    this.bal = this.bal + a; }

    public void addInterest()
     { int interest = this.bal * this.interestRate/100;
          this.deposit(interest);  }
}

class CurrentAccount
{  private int chargeRate;
   private TransactionHistory history;

   public deposit(int a)
    {  this.bal = this.bal + a;
       TransactionHistory h  = new TransactionHistory();
        h.amount = a;
        h.history = this.history;
        this.history = h; }
}

class  TransactionHistory
{    private int       amount;
     private TransactionHistory  history;
}
```

Figure 3. Implementation model

2.2 The Implementation Model

An implementation model is given by an implementation subsystem (in the sense of [10]) which contains components that may be related by dependency relations. In this paper, any component `C.java` will be a Java file containing the code of a Java class `C`. It is assumed that all attributes of Java classes are declared “private” or “protected” to ensure the encapsulation of object states. The code of the implementation model is shown in Figure 3.

2.3 The Realization Relation

A design model and its corresponding implementation model can be related by realization relation as shown in Figure 4. We say that the realization relation between “designSubsystem” and “javaSubsystem” holds if the following syntactic and semantics requirements are satisfied:

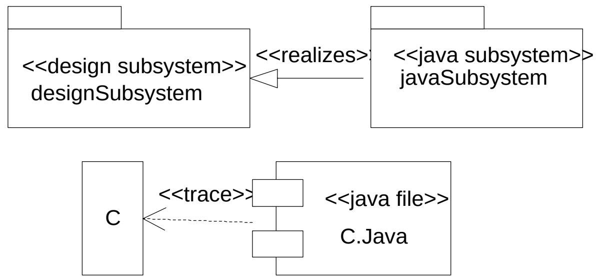

Figure 4. Trace dependency and realization relation

Syntactic requirements

Every class in the design subsystem will have a corresponding component in the implementation subsystem. This can be achieved by using trace dependencies as considered in [10]. It is required that every class `C` of the design model is related by a trace dependency to a Java component `C.Java` as shown in Figure 4. The trace dependency between a class `C` and `C.java` is supposed to hold if and only if the following conditions are satisfied:

a. Each attribute of the design class `C` is also an attribute of the Java class `C` and for each role name at the end of a directed association the Java class contains a corresponding reference attribute with the same name. Standard types may be slightly renamed according to the Java syntax and that role names with multiplicity greater than one map to a reference attribute of some container type.

b. For each operation `m` specified in the design class `C` there is a method declaration in the Java class `C` and vice versa (up to an obvious syntactic modification of the signature). The operation `m` of the design model has the property {abstract} if and only if the method `m` is an abstract method.

c. The design class `C` is a (direct) subclass of a design class `A` if and only if the Java class `C` extends the Java class `A`.

These conditions guarantee (in particular) that the OCL expressions used as constraints for the design model can be interpreted in the implementation model which is necessary to define the semantical requirements. Moreover, note that the above conditions are satisfied by usual code generators for Java classes from UML class diagrams.

Semantic requirements

For the formulation of the semantic requirements we assume that the syntactic requirements from the above are satisfied. Given that, we require that for each design class `C` the following are satisfied:

1. For each operation of `C` its precondition and postcondition are respected by corresponding method implementations. That is for each operation `m` specified in the design class with OCL-constraints:

```
context C::m(p1: T1, ..., pn : Tn)
pre: OCL-PRE
post: OCL-POST
```

the given Java subsystem has the following specification for the method `m`:

```
/*@  private normal_behavior
  @  requires JML-PRE;
  @  assignable variable-List;
  @  ensures JML-POST;
  @*/
void C::m( T1 p1, ..., Tn pn)
```

where `JML-PRE` and `JML-POST` are the interpretations of the precondition OCL-PRE and postcondition OCL-POST in JML respectively. The first line of the JML specification indicates that the specification is private and that the method terminates normally (no exception) under the assumption that the precondition holds. The specification is private because private variables are used, and the normal behaviour is because OCL specification does not specify exceptions. However, the specification could be designated public where the private variables are regarded as public for specification purposes. JML preconditions follow the keyword “`requires`”, and postconditions follow the keyword “`ensures`”. The assignable clause indicates which variables or attributes the method is allowed to modify. OCL does not support the specification of frame conditions so this clause has to be extracted from the postcondition.

If in the design model, there is a subclass `C'` of the design class `C` which redefines `m` in the sense that it provides an additional OCL constraint with precondition PRE' and postcondition POST' for `m`, then the method `m` must satisfy both of the following:

```
/*@ private normal_behavior
  @ requires JML-PRE;
  @ assignable variable-List;
  @ ensures JML-POST;
  @*/
void C::m( T1 p1, ..., Tn pn)
```

```
/*@ also
  @ private normal_behavior;
  @ requires JML-PRE';
  @ assignable variable-List;
  @ ensures JML-POST';
  @*/
void C'::m( T1 p1, ..., Tn pn)
```

That is the specification of the method in the superclass is inherited. In JML this is indicated by adding the keyword '`also`' before the specification in the subclass. This ensures that Liskov's substitution principle is satisfied. The implementation system given with JML specification is shown in Figure 5.

2. Each invariant attached to a class in the design model will be translated to a JML invariant on the corresponding class in the implementation subsystem. These invariants are preserved by method implementations. This means that for each invariant OCL-INV for a design class C, there is a corresponding invariant `JML-INV` for the java class `C` expressed in JML. Subclasses inherit the invariants of their superclasses.

```
class abstract BankAccount {
     private int bal;

     /*@    private normal_behavior
       @    requires   a >= 0;
       @    assignable bal;
       @    ensures    bal = \old( bal)  + a;
       @*/
     void deposit( int a);
}
class DepositAccount extends BankAccount {
     private int interestRate;
     //@ invariant  bal >= 0 && interestRate >=0;

     public void deposit(int a) {
            this.bal = this.bal + a;
     }
     /*@   private normal_behavior
       @   assignable bal;
       @   ensures    bal = \old(bal)  +
                          \old(bal)*interestRate* 0.1;
       @*/
     public void addInterest(){
        private int interest = bal * interestRate/100;
         this.deposit(interest);
     }
}

class  CurrentAccount extends BankAccount {
     private    int   chargeRate;
     private    TransactionHistory  history;

     //@ invariant chargeRate >= 0;

    /*@   private normal_behavior
      @   requires  a >= 0;
      @   assignable bal, history;
      @   ensures    \fresh( history) &&
      @              history.amount = a &&
      @              history.history = \old(history);
      @*/
     public void deposit(int a){
          this.bal = this.bal + a;
          TransactionHistory h  =
                       new TransactionHistory();
          h.amount = a;
          h.history = this.history;
          this.history = h;
     {
```

Figure 5. Implementation subsystem with JML specification

2.4 Mapping OCL Expressions

2.4.1 Boolean expressions

OCL assertions are boolean expressions built using the boolean operators and (conjunction), or (disjunction), not (negation), implies (implication), forAll (universal quantification), and exists (existential quantification). The basic boolean expressions are mapped quite easily. That is the assertions (p and q) is mapped to the JML expression `p&&q`, where p and q are the translation of the expressions p and q respectively. Since assertions are expressions, an assertion may be undefined. The OCL's semantics for undefinedness is to allow an undefined value to stand for undefined expressions resulting in three-valued logic interpretation. So if p is false, the expression (p and q) will evaluate to false regardless of what the value of q is. This would not be consistent with the Java interpretation of `p&&q` sine the order of evaluation is important. However, the way JML deals with undefinedness is to substitute an arbitrary but defined value for an undefined expression. So if p is false, the expression `p&&q` will evaluate to false regardless of what the value of q is. So the mapping is still sound. In the same way, the expressions p or q, not(p), and p implies q are mapped to `p||q`,`!p`, and `p==>q` respectively.

The operation forAll is applied to finite collections (sets, bags, sequences) with the syntax c->forAll(x : T | p(x)), where c is a collection containing elements of type T, and p is a predicate with x as parameter. This expression asserts that all the elements of c satisfy the predicate p. It is mapped to the JML assertion `(\forAll T x;c.has(x);p(x))`, where `c.has(x)` is the membership predicate for collection c. The exists operation is used to assert that at least one element of a collection satisfies a given predicate. Its syntax is given as c->exists(x : T | p(x)), where c is a collection containing elements of type T, and p is a predicate with x as parameter. It is mapped to the expression `(\exists T x;c.has(x);p(x))`.

2.4.2 Collection types

OCL supports the following collection types Set(T), Bag(T), and Sequence(T). All the mathematical collections in JML are supported in the form of Java immutable or pure classes. The class `JMLValueSet` is the class of objects representing sets which contain values (immutable objects). While the class `JMLObjectSet` is the class of objects representing sets which contain other objects. This distinction is needed in order to define equality for sets containing values and sets containing objects. Other collection types are also defined by classes containing collections with values and classes containing collections with objects. These include bags (`JMLValueBag`, `JMLObjectBag`) and sequences (`JMLValueSequence`, `JMLObjectSequence`).

When mapping expressions involving collections it is essential to consider whether the collection containing values or objects, since these are treated differently in JML. For instance, the OCL type Set(T), where T is a value type, will be mapped to a JML set type containing values of type T. Since JML collection types hold elements of the type Object (the super class of Java classes), Set(T) is mapped to JMLValueSet. However, a constraint is needed to say that the elements belong to T. For example, if s is of type Set(T), then the s is mapped to a set s of type JMValueSet which satisfies the condition (\forall JMLType x;;s.has(x)==> x instanceof T).

The type Set(Integer) is mapped to JMLValueSet, where each set contains objects of the type JMLInteger rather than of type int. This is because JML collections contains objects not basic values, where some of these objects represent values (i.e. immutable objects). If T is an object type, Set(T) is mapped to the type JMLObjectSet, the type of sets containing object references.

OCL collections contain basic values as well as objects and "=" is used to denote equality between collections. For instance, if s1 and s2 are of type Set(T), s1=s2 asserts that s1 and s2 are equal in the sense that they contain the same elements. Since JML collection types are defined as Java *immutable* (or pure) classes, using == as the equality between collections will give wrong results in the presence of the New operator. Instead, the equality used is a defined one, namely equals. So the assertion s1=s2 is mapped to s1.equals(s2).

The standard OCL operations on collections can be translated quite easily since most of them are directly supported by JML. In addition OCL also supports operations for filtering collections. For example, the operation select filters a finite collection using a boolean predicate. Its syntax is of the form c->select(x : T | p(x)), where c is a finite collection with elements of type T and p is a boolean predicate. The value of this expressions is the subcollection of c obtained by taking those elements satisfying predicate p. JML collections do not have a corresponding operation for filtering a finite collection, however a form of comprehension can be used to map expressions involving the operation select. So is the expression s->select(x : T | p(x)) where s is a set and T is an object type is translated to new JMLObjectSet{ T x | s.has(x)&&p(x)}. JML only supports set comprehension, that is bag and sequence comprehension is not supported. OCL has operations that convert one type of collection into another. For example, asSet converts a bag or sequence to a set. There are no corresponding methods in JML for such operations, however, these can be defined within JML by adding methods to the collection classes.

The operation collect maps a collection into another collection based on an expression. Its general syntax is given by c->collect(x : T | expr(x)), where c is a collection of T's elements and expr is an expression that may involve variable x. The value of this expression is another collection obtained from c by applying expr to each element of c. JML does not have a corresponding operation. However, one could extend JML with a general form of comprehension such that new JMLObjectSet{S expr(x)|s.has(x)} would denote the set obtained from a given set s by applying expr to each element in s.

2.4.3 Old values and oclIsNew

OCL uses the @ symbol followed by the pre keyword to refer to the pre-state value of an attribute or association. That is, bal@pre in the postcondition of deposit refers to the value of the attribute bal in the pre-state. JML uses \old to refer to the pre-state value of a variable with the syntax \old(e), where e is an expression. The meaning of \old(e) is as if e were evaluated in the pre-state and that value is used in place of \old(e) in the assertion. Expressions of the form expr@pre are mapped to \old(expr).

When navigating optional associations on a class diagram, the value of navigation expression might be undefined. This is the case when an object from one end of the association is not related to an object on the other end of the association with "0..1" multiplicity. In order to check if an expression is defined OCL treats it as a set. For example acc.history is defined if and only if acc.history->notEmpty. The expression a.history->notEmpty is then mapped to a.history != null.

The OCL operation oclIsNew is used in postconditions to assert that an object is newly created. In JML, the operator \fresh is used for similar purpose. That is \fresh(o) asserts that the object bound to o was not allocated in the pre-state. The expression o.oclIsNew is mapped to \fresh(o).

3. Verifying the Realization Relation

In this section, we show how a proof of correctness of the realization relation of the AccountSubsystem (Figure 6) could be done. According to the definition in Section 2.3, one needs to show the trace dependencies, the satisfaction of the preconditions and postconditions constraints and the preservation of OCL invariants.

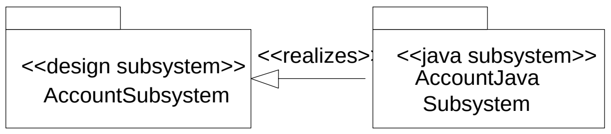

Figure 6. Realization relation of the account subsystem

Trace dependencies what needs to be shown is that for each class of the AccountSubsystem there exists a corresponding Java class in AccountJavaSubsystem such that the attributes, methods, and generalization relations are preserved. By looking at the design and implementation subsystems, the trace dependencies are satisfied.

Satisfaction of preconditions/postconditions The JML method specifications generated from the OCL operation specifications have to be verified. It is easy to see that the implementation of `deposit` in the class `DepositAccount` satisfies the postcondition. It is possible to use the tools developed by the LOOP project [7] which provide a verification platform for Java programs based on the formal semantics of both Java and JML which has been given in [13]. This approach is based on translating a method specification and implementation to proof obligations that can be verified using a suitable theorem prover such as PVS.

Preservation of invariants The AccountSubsystem has two invariants for `DepositAccount` and `CurrentAccount`. From the implementation subsystem it is easy to show that the invariants are preserved. To show that the method `deposit` of the class `DepositAccount` preserves the invariant one needs to prove:

```
old(bal)>=0 && old(interestRate)>=0 && (a>=0)
            ==>(bal>=0)
```

Since the implementation of `deposit` meets its postcondition, the following is true `bal=old(bal)+a`. From this it follows that `bal>=0`. Since the method does not change the value of `interestRate`, it follows that `interestRate>=0`.

4. Conclusion

The realization of UML design models by Java subsystems has been formalised in the context of the JML. This is done by mapping expressions and assertions written in OCL to JML expressions and assertions. The formalization provides a formal approach for verifying the realization relation where JML is used as the assertion language. One possible way to do the verification is to use the tools developed by the LOOP project which translates Java code annotated with JML specifications into PVS assertions and proof obligations. These obligations could then be verified using PVS. This approach can also be used in the context of CASE tools which generate Java code from UML diagrams, where OCL constraints can also be mapped to JML assertions. This will help in debugging and testing using the JML runtime assertion checker to check invariants, preconditions and postconditions of methods.

One area of future work is to extend this approach to include interfaces, since the design models in this paper were restricted to classes and their relationships.

References

[1] K. Arnold and J. Gosling, *The Java programmingLanguage*. The Java Series. Addisson-Wesley, Reading, MA, second edition, 1998.

[2] A. Borgida, J. Mylopoulos, and R. Reiter, On the frame problem in procedure specifications. IEEE Transactions on Software Engineering, 21(10): 785-798, October, 1995.

[3] Y. Cheon and G. Leavens, A Runtime Assertion Checker for the Java Modeling Language (JML). Technical report TR#02-05a, April 2002 revision, Department of Computer Science, Iowa State University.

[4] D. D'Souza and A. Wills, Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1998.

[5] B. Jones, Systematic Software Development using VDM. Prentice Hall 1990.

[6] G. Leavens, A. Baker and C. Ruby, Preliminary Design of JML: A Behavioral Interface Specification Language for Java. TR98-06, revised version 2002.

[7] B. Jacobs et al., The LOOP project: Formal methods for object-oriented systems at http://www.cs.kun.nl/~bart/LOOP.

[8] Rational Software Corporation. The Object Constraint Language Specification Version 1.4. Available from http://www.rational.com, 2001.

[9] Rational Software Corporation. The Unified Modeling Language Version 1.4. Available from http://www.rational.com, 2001.

[10] J. Rumbaugh, I. Jacobson and G. Booch, The Unified Modelling Language Reference manual. Addison-Wesley, Reading, Mass., 1998.

[11] J. Spivey, The Z Notation: A Reference Manual. International Series in Computer Science. Prentice-Hall, New York, N. Y., second edition, 1992.

[12] J. Warmer and A. Kleppe, The Object Constraint Language: Precise Modelling with UML. Addison-Wesley, 1999.

[13] E. Poll and B. Jacobs, A Logic for the Java Modelling Language JML. Computing Science Institute Nijmegen, Technical Report CSI-R0018. University of Nijmegen, 2001.

[14] The PVS Specification and Verification System at *http://pvs.csl.sri.com/*.