# Pattern-Based Mapping of OCL Specifications to JML Contracts

Ali Hamie
*Computing Division, Brighton University, Brighton, UK*
*a.a.hamie@brighton.ac.uk*

Keywords: Constraint Pattern:OCL:JML:Contracts

Abstract: OCL is a formal notation to specify constraints on UML models that cannot otherwise be expressed using diagrammatic notations such as class diagrams. The type of constraints that can be expressed using OCL include class invariants and operation preconditions and postconditions. However, since OCL constraints cannot be directly executed and checked at runtime by an implementation, constraints violations may not be detected or noticed causing many potential development and maintenance problems. This paper describes an approach for deriving a JML specification for a java implementation (a contract) from a specification written in UML and augmented with OCL constraints. This facilitates the checking of OCL constraints at runtime by translating them to JML executable assertions. The approach is based on the concept of constraint patterns that enable the reuse of commonly occurring constraints within a given context in software modelling. Each OCL constraint pattern would be mapped to a corresponding JML pattern that can be used in the translation process. The result is a library of JML constraint patterns that provides a seamless transition from UML/OCL designs to Java implementations.

## 1 INTRODUCTION

The Unified Modeling Language (UML) (OMG, 2006) has been widely accepted as the standard object-oriented modelling language and is supported by a large number of CASE tools. The Object Constraint Language (OCL) (Warmer and Kleppe, 2003) is an integral part of UML, and was introduced to express additional constraints on models that diagrams cannot convey by themselves. For example a UML diagram such as class diagram cannot express all the relevant constraints about the application being modelled. So class models must typically be refined with textual constraints written in OCL. These constraints include invariants on classes, and preconditions and postconditions of operations. The combination of UML/OCL in modelling results in more precise and abstract models. However, developing constraint specifications is not an easy task. Among other things, one important aspect needs to be taken into account: class diagrams can express complicated relationships, including subtyping, reflexive relations, or potentially infinitely large instances, and constraining such facts requires dealing with this complexity. In order to facilitate and simplify the development of constraints, the concept of specification patterns has been introduced as *constraint patterns* in MDE (Ackermann and Turowski, 2006; Wahler et al., 2006; Davis and Bonnell, 2007). A constraint pattern captures and generalizes frequently used logical expressions. It is a parameterizable constraint expression that can be instantiated to solve a class of specification problems. At a more formal level, a constraint pattern with respect to a meta-model can be defined as a function that maps a set of meta-model elements to a constraint.

As a design notation, however, OCL is not executable and OCL constraints are not reified to implementation artifacts. This could lead to development and maintenance problems of constraints such as inconsistency. One way to overcome some of these problems is to map OCL constraints to source code in a form that can be executed and checked at runtime. Hamie (Hamie, 2004) has defined rules for translating OCL expressions and constraints to JML expressions and assertions. This translation was refined and implemented in (Avila et al., 2008) by providing a JML class libraries for OCL collection types that simplifies the translation. JML is a behavioural interface specification language that can be used to specify Java classes and interfaces(Leavens et al., 2006), and a significant subset of it can be can be checked at runtime (Cheon et al., 2002). JML is very specific to the programming language Java and thus handles many low-level details. What makes JML suitable for the trans-

lation is that it supports several language and tool features, in particular, specification only variables called *model* variables (Cheon et al., 2005) and specification refinements.

Building on previous work, this paper proposes an approach for translating a UML/OCL specification to a specification for a Java implementation (a *contract*) based on constraint patterns. The main components of this approach are a set of JML constraint patterns that can be used for the translation of OCL constraints to JML assertions. Each OCL constraint pattern is translated to a JML specification pattern described as a JML template. The use of constraint patterns makes the translation intuitive and traceable. It is also expected that the use of patterns facilitate automation of the translation.

In addition of being used for the translation of constraints, the JML constraint patterns are inspired by those patterns used in OCL, and as such they are useful for simplifying the development of assertions for Java classes and interfaces. This is important since assertions are recognised as a practical programming tool and are said to be more effective when derived from formal specifications such as OCL constraints.

The remainder of the paper is organised as follows. In Section 2 we briefly review OCL, specification patterns, and JML through a small example that will be used throughout the paper. In Section 3 we describe a way for translating an OCL constraint pattern to a JML pattern. In Section 4 we apply the approach to some OCL constraint patterns and our running example. Section 5 provides the conclusion and future work.

# 2 BACKGROUND

## 2.1 Object Constraint Language

The Object Constraint Language (OCL)(Warmer and Kleppe, 2003) is a textual, declarative notation that can be used to specify constraints or rules that apply to UML models. OCL can play an important role in model driven software engineering because UML class diagrams are not precise enough to enable the transformation of a UML model to complete code. In fact, it is an important component of OMG's standard for model transformation for the model-driven architecture (Frankel, 2002).

A UML class diagram alone cannot express all relevant constraints about an application. The diagram in Figure 1, for example, is a UML class diagram modelling a video rental store. There are various additional constraints on the model that cannot be

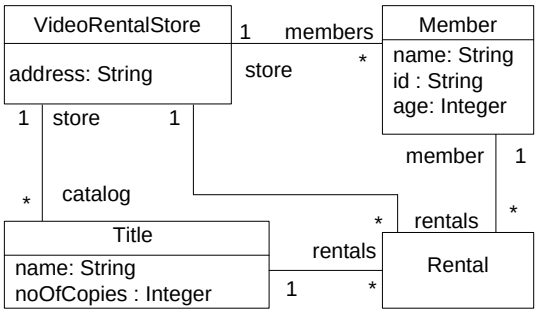


Figure 1: A partial class model for video rental store.

expressed diagrammatically. For example, a member can only rent one copy of a video title at one particular time or the number of copies of a title is greater than zero. It is very likely that a system based only on diagrams alone will be incorrect. Such additional constraints can be precisely described using the OCL which is based on predicate logic and mathematical set theory. For example a simple constraint stating that the number of copies of a title is greater than zero can be expressed as follows.

---

**context** Title

**inv** : copiesGreaterZero : **self**.noOfCopies $> 0$

---

This constraint, called an *invariant*, states a fact that should be always true in the model. The actual invariant is represented as an OCL boolean expression using the variable self that refers to an object of class Title. copiesGreaterZero is the name of the invariant.

It is also possible to use OCL in order to specify the behaviour of an operation. For example, the following OCL constraints specifies the behaviour of an operation Title::addCopies(n : Integer) using a pair of predicates describing a precondition and a postcondition.

---

**context** Title::addCopies(n: Integer)

**pre** : n $> 0$

**post** : noOfCopies = noOfCopies@pre + n

---

The pre and postconditions state that if invoked with parameter n greater than zero the operation sets the new number of copies of the title by adding n to the previous number of copies. In the postcondition, the @pre annotation denotes the value of a property at the precondition time.

OCL supports several primitives types such as Integer, Real, Boolean, and String and collection types such as Collection, Set, OrderedSet, Bag, and Sequence (Warmer and Kleppe, 2003). These types are equipped with various operations that can be used for

writing OCL constraints. For example the collection operation size returns the number of elements contained in a collection, and the forAll operation checks whether an expression is true for all objects in a given collection.

## 2.2 Specification Patterns in OCL

Constraint patterns have been introduced in OCL to simplify and speed up the development of constraints and to ensure their consistency (Ackermann and Turowski, 2006; Wahler et al., 2006; Davis and Bonnell, 2007). Common constraint expressions are generalised and captured as constraint patterns. These patterns can then be instantiated in specific contexts to generate the concrete constraints.

An example of a constraint pattern is the Attribute Value Restriction pattern, which can be used to restrict the attribute values of a class. It has three parameters: property of type Property, denoting an OCL property, operator and value of type OclExpression, denoting an OCL operator and an expression respectively. This pattern is described in OCL by the following template.

---

**pattern** AttributeValueRestriction(property: Property,
operator: OclExpression, value: OclExpression) =
**self**.property operator value

---

The *Attribute Value Restriction* pattern can be applied by providing actual values for the formal parameters. For instance, the constraint that each member of the video store is over 18 can be stated as the following invariant.

---

**context** Member
**inv** over18 : AttributeValueRestriction(age, $>$, 18)

---

The pattern has been applied in the context of the class Member where the parameters property, operator and value have been replaced by the values age, $>$ and 18 respectively. By unfolding the above constraint one obtains the following invariant.

---

**context** Member
**inv** over18 : self.age $>$ 18

---

A good tool can show an unfolded version of the constraint on demand.

In (Wahler et al., 2006) an extensible library of *elementary* constraint patterns was presented for OCL modelling. The idea of elementary constraint patterns is to identify a relevant set of atomic constraints that covers frequently occurring restrictions on a model, e.g. restrictions on attribute values or on relations between objects. In addition to elementary constraint patterns, *composite* constraint patterns were introduced in order to express complex properties by combining an arbitrary number of other constraints. The identified composite patterns include Negation, If-Then-Else, Exists, Or, and And. Tool support for specification patterns is provided in the form of a set of plug-ins for the MDE tool IBM Rational Software Architect (RSA) that enable consistency-preserving refinement of UML class models with constraint patterns (Wahler et al., 2006).

## 2.3 Java Contracts in JML

The Java Modeling Language (JML) (Leavens et al., 2006) is a formal specification language that can be used to specify Java classes and interfaces. As such JML provides an extended Design-by-Contract concept to the programming language Java. The Design-by-Contract (DBC) concept includes conventional clauses for preconditions and postconditions of methods as well as class invariants. JML specifications or assertions can be added directly to source code as a special kind of comments called *annotation comments*, or they can live in separate specification files. These assertions are usually written in a form that can be compiled, so that their violations can be detected at runtime. In addition, JML provides clauses for specifying exceptions, and its extensions include model and ghost variables which describe specifications only data and therefore allow the modelling of abstract state space. The relationship between the concrete state space and the abstract one is achieved by the use of 'represents' clauses in the concrete classes and thus formulate a data refinement relation.

```
// File: Title.java
public class Title {
   /*@ spec_public @*/ private int noOfCopies;
   //@ public invariant  this.noOfCopies >= 0;

   /*@ requires  n >= 0;
     @ assignable noOfCopies;
     @ ensures noOfCopies == \old(noOfCopies) + n
     @*/
   public void addCopies(int n) {}

   // the rest of definition
}
```

Figure 2: Sample Java code with JML annotations.

Figure 2 shows a sample Java code annotated with JML specification written in a Java source (.java) file. The annotation comments in the source code are indicated by //@ and /*@ ... @*/. It describes

the behaviour of class `Title`. The JML keyword *spec−public* states that the private field `noOfCopies` is treated as public for specification purpose; e.g., it can be used in the specifications of public methods such as `addCopies`. The example also shows the specification of the method `addCopies`. A method specification precedes the declaration of the method. The *requires* clause specifies the precondition, the *assignable* clause specifies the frame condition, and the *ensures* clause specifies the postcondition. The JML keyword *old* in the postcondition denotes the pre-state value of its expression; it is mainly used in the specification of a method such as `addCopies` that changes the state of an object.

JML is widely accepted and supported by a range of tools covering the different levels of program verification from runtime checking (Iowa State JML tools, via (Leavens et al., 2003)) to static checking (ESC/Java2, (Cok and Kiniry, 2004)) to 'real' interactive verification (Loop project, (Jacobs and Poll, 2003)). Typically JML extensions are encapsulated in specially formatted Java comments, so that any Java tool can still handle the source code.

# 3 Mapping OCL Patterns to JML

A UML/OCL specification can be mapped to JML by identifying constructs in the Java program that match the modelling elements of UML/OCL. These include types, expressions, operations and classes.

The Types in OCL are directly mapped to Java types, i.e. Java interfaces providing the required methods. For certain integer subsets that fit in Java′s `int` range we will use that, otherwise we map them also to interfaces that encapsulate JML′s `\bigint`. Sets, relations, etc. can be mapped to JMLs model classes, but we have to perform cast operations for retrieving the original types since JML does not yet support Java′s generics.

OCL expressions are rephrased as Java expressions. Predicates in OCL are boolean expressions and mapped to Java boolean expressions extended with various forms of quantifications including universal and existential quantifications.

In order to translate OCL constraint patterns to JML patterns we make use of previous work done on translating OCL into JML. For instance (Hamie, 2004) defines rules for mapping OCL expressions to JML expressions. A library-based approach for translating OCL to JML is given in (Avila et al., 2008). The latter approach aims at making the mapping easier by building a library of Java classes for the OCL types and expressions. The current approach builds on previous work and aims to make the translation more intuitive and traceable.

## 3.1 Constraint Patterns in JML

The key idea of our approach is to introduce a set of constraint patterns for JML corresponding to OCL constraint patterns. That is the identified JML patterns are directly inspired by OCL patterns. The semantics of JML constraint patterns can be captured as a JML template, i.e. a parameterizable JML expression. To be more specific, the template can be used as *macros* because patterns are untyped. The syntax of an JML template starts with the keyword pattern followed by the name of the pattern and a set of typed parameters in brackets. This is followed by an equals sign and an arbitrary JML expression in which the name of formal parameters can be used. This is similar to the syntax of OCL templates. A JML pattern can be defined as a function that maps a set of metamodel elements to a Java constraint. In the following, we define the *Singleton* design pattern using the template language, assuming that there is a method `getInstances` that returns the set of instances of a Java class.

```
pattern Singleton(element:Class) =
    element.getInstances().size() == 1
```

JML templates can be used as first-class language elements of JML. When they are instantiated, the formal parameters are replaced by the values of the actual parameters. As an example, we instantiate the *Singleton* pattern to constrain the number of `VideoRentalStore` objects in a model state to one. Note that in the following code the two listed invariants of the class `VideoRentalStore` are semantically equivalent.

```
public class VideoRentalStore {
  //@ invariant  Singleton(VideoRentalStore);
  /*@ invariant
    @ VideoRentalStore.getInstances().size()==1;
    @*/
}
```

The above example shows that constraint patterns are concise means of hiding the syntactic and semantic complexity of JML expressions and offering a unique name and uniform interface to the programmer.

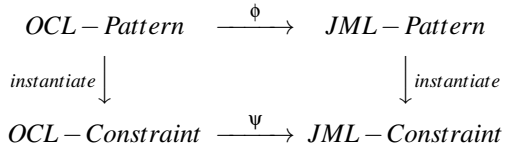


Figure 3: Constraint pattern mapping diagram.

## 3.2 Translating OCL Patterns

The general form of an OCL constraint pattern is given by the following template.

---

**pattern** patternName($p_1 : Type_1, ..., p_n : Type_n$) = patternBody

---

patternName stands for the name of the pattern, $p_1, ..., p_n$ is the list of parameters for the pattern of types $Type_1, ..., Type_n$ respectively, and patternBody is the body of the pattern represented as an OCL expression built using the parameters and OCL operations. The types of parameters $Type_1, ..., Type_n$ are types from the UML/OCL metamodel.

The corresponding JML pattern is obtained from the OCL pattern by mapping the types of parameters to types in the JML metmodel, and by translating the body of the pattern into a JML expression. If $Type_i$ is mapped to $jmlType_i$ $(i = 1, .., n)$ and patternBody is mapped to jmlpatternBody then the JML pattern is given as follows.

---

**pattern** oclPatternName($p_1 : jmlType_1, ..., p_n : jmlType_n$) =
jmlpatternBody

---

Translating the body of the OCL pattern to a body for the corresponding JML pattern can be achieved by using the translation rules defined in (Hamie, 2004; Avila et al., 2008). This process can be applied to each OCL constraint pattern. The diagram in Figure 3 can be interpreted as saying that there are two ways to obtain a JML assertion from an OCL constraint. The starting point is applying an OCL constraint pattern. Unfolding the definition of the pattern we obtain the OCL constraint which can be mapped to a JML assertion using the mapping defined in (Hamie, 2004). The other way is not to unfold the OCL pattern but to use the corresponding JML pattern to do the mapping. Then the JML assertion can be obtained by unfolding the definition of the JML pattern. Assuming appropriate mappings have been used, the two ways should lead to semantically equivalent JML assertions.

# 4 APPLYING THE APPROACH

In this section we apply our approach to the video rental store model. This will be based on some existing OCL constraint patterns as introduced in (Wahler et al., 2006).

## 4.1 Attribute Value Restriction

In background section (Section 2.1) on OCL, we introduced the constraint copiesGreaterZero, which we defined as follows.

---

**context** Title
**inv** : copiesGreaterZero : **self**.noOfCopies $> 0$

---

This constraint represents a common kind of constraint, namely simple value restrictions for attributes. So the *Attribute Value Restriction* pattern was introduced in order to restrict the values of attributes for all instances of the attributes' class. This pattern was introduced in the background section (Section 2.2) of specification patterns, which is defined by the following OCL template.

---

**pattern** AttributeValueRestriction(property: Property,
operator: OclExpression, value: OclExpression) =
**self**.property operator value

---

Translating the *Attribute Value Restriction* pattern to a JML pattern involves mapping the parameters of the OCL pattern into parameters of the JML pattern with appropriate types. This assumes we have in place metamodels for both OCL and JML. In this case property is mapped to a JML field of the same name and with type Field. The parameters operator and value are mapped to JML expressions with the same names of type JmlExpression. This followed by mapping the body of the pattern which an OCL expression. In this case the expression self.property operator value is simply mapped to `this.property operator value`. Therefore we obtain the following JML pattern.

---

**pattern** oclAttributeValueRestriction(property: Field,
operator: JmlExpression, value: JmlExpression) =
**this**.property operator value

---

In section 2.2 we applied the *Attribute Value Restriction* pattern in the context of the UML class Member and obtained the following invariant.

```
context Member
inv over18 : AttributeValueRestriction(age, >, 18)
```

Before translating this constraint to a JML assertion, we need to implement the class Member and its attributes in Java. In this case we map Member to a Java class with the same name and map the attribute age to a Java field *age*. The invariant of the class Member is then translated by applying the corresponding JML pattern with actual parameters `age`, $>$ and 18. The class `Member` with the translated invariant is given as follows.

```
public class Member {
  /*@ spec_public @*/ private int age;
  /*@ public invariant
      oclAttributeValueRestriction(age, >, 18);
    @*/
}
```

By unfolding the pattern definition, we obtain the following JML invariant:

```
public class Member {
   /*@ spec_public @*/ private int age;
   //@ public invariant  this.age > 18
}
```

## 4.2 Multiplicity Restriction

The multiplicities of properties (associations) can only be roughly constrained in a diagrammatical way in class models. However, there are situation where the multiplicity of an association depends on the value of an attribute. For example, an object of class Title can have an arbitrary number of rentals which cannot exceed the total number of copies for that title. So there is a dependency between the association with role name rentals and the attribute noOfCopies. Here we are assuming that the model deals with current rentals rather than past rentals. A constraint pattern named *Multiplicity Restriction* is defined in (Wahler et al., 2006) to capture this kind of constraints. This pattern is presented as an OCL template as follows.

```
pattern MultiplicityRestriction (navigation : Sequence(Property),
        operator: OclExpression, value: OclExpression) =
self.navigation->asSet()->size() operator value
```

This pattern has three parameters: navigation, represented as a sequence of properties, thus allowing the use of OCL navigation expressions such as self.catalog.rentals, operator, and value, which can be arbitrary OCL expressions. value can be the name of an attribute or an arbitrary OCL expression. Since self.navigation may result in a bag, the OCL operator asSet() is used to convert the resulting collection into a set.

Using the *Multiplicity Restriction* pattern, we can define the constraint RentalsRestriction as follows.

```
context Title inv: RentalsRestriction :
MultiplicityRestriction(rentals, <=, noOfCopies)
```

This is done by replacing the parameters navigation, operator and value by rentals, $<=$, and noOfCopies respectively.

The *Multiplicity Restriction* pattern can be translated to the following JML pattern.

```
pattern oclMultiplicityRestriction (navigation:List<Field>,
        operator: JmlExpression, value: JmlExpression) =
this.navigation.uniqueSet().size() operator value
```

The expression `this.navigation` represents the translation of the OCL navigation expression self.navigation built from the sequence of properties given as a parameter. Since the result of `self.navigation` is a bag, we used the Java method `uniqueSet` to convert a bag into a set.

The translation of the multiplicity restriction invariant for *rentals* property is given in the following code.

```
public class Title {
  /*@ spec_public @*/ private int noOfCopies;
  /*@ spec_public @*/ private Set<Rental> rentals;
  /*@ public invariant
      oclMultiplicityRestriction(rentals,
                                 <=, noOfCopies);
    @*/
}
```

The implementation of the class Title uses JDK set with its multiplicity expressed as a class invariant. The translation in this particular case uses program variables (i.e. the `rentals` field) in the assertion. One of the potential problems with this approach is the maintenance of both OCL constraints and Java programs. For example, changing the implementation from sets to arrays will affect the whole JML assertion. This might make it necessary to rewrite the JML

assertions in terms of the vocabulary of the new representation, i.e. arrays. One way to overcome this problem is to use *model fields* (Cheon et al., 2005) which are specification only variables. That is they can be referred to only in assertions, but not in program code. The pattern-based approach is useful maintaining assertions since changing the body of a pattern would apply to every assertion defined in terms of that pattern.

By unfolding the above invariant we get:

```
public class Title {
  /*@ spec_public @*/ private int noOfCopies;
  /*@ spec_public @*/ private Set<Rental> rentals;
  /*@ public invariant
        this.rentals.size()<= noOfCopies;
    @*/
}
```

## 4.3 Unique Identification

The *Unique Identification* pattern is very frequent. For example, in the video rental model it is required that the id for members is unique. That is any members m1 and m2 should be distinguishable by their membership identities. In OCL such constraint is expressed as follows.

```
context Member
inv UniqueID: Member.allInstances->isUnique(id)
```

This constraint can be generalized to composite primary keys by using the OCL tuple type.

The *Unique Identifier* pattern (Wahler et al., 2006) (referred to *Semantic Key* in (Ackermann and Turowski, 2006)) captures the situation where an attribute (or a group of attributes) of a class plays the role of an identifier for the class. That is the instances of the class should differ in their values for that attribute (group). The corresponding OCL template is given as follows.

```
pattern UniqueIdentifier (class: Class, property : Property) =
class.allInstances->isUnique(property)
```

This pattern has two parameters class which represents the context of the invariant, property, which denotes a property that have to be unique for each object of the context class. The body of the pattern is defined in terms of the operation allInstances, which returns the set of existing instances of the class, and the operation isUnique. This pattern can be generalised to more than one property by using the OCL tuple type.

The corresponding JML pattern can be defined as follows.

```
pattern oclUniqueIdentifier(class:Class,property:Field) =
    (\forall class a1, a2;\created(a1) && \created(a2);
          a1 != a2 ==> a1.property != a2.property)
```

The pattern *Unique Identifier* takes a class and a property as parameters. It asserts that for any two distinct created instances `a1` and `a2` of class `class`, their values for `property` are different. The JML keyword `\created` restricts the range of the quantification to created objects.

To make the correspondence between OCL and JML simpler it is possible to introduce a primitive quantifier (`\unique` that asserts the uniqueness of a property or field. In that case the *Unique Identifier* pattern can be concisely stated as follows.

```
pattern oclUniqueIdentifier(class:Class, property:Field) =
    (\unique class a;\created(a);property)
```

Applying the *Unique Identifier* pattern in the context of the class Member we get the following invariant.

```
context Member
inv UniqueID: UniqueIdentifier(Member, id)
```

The following code shows the translated invariant in JML.

```
public class Member {
  /*@ spec_public @*/ private int id;
 /*@ public invariant
       oclUniqueIdentifier(Member, id);
   @*/
}
```

By unfolding the above invariant we obtain:

```
public class Member {
  /*@ spec_public @*/ private int id;
 /*@ public invariant
       (\forall Member a1, a2;
             \created(a1) && \created(a2);
                  a1!=a2 ==> a1.id != a2.id);
   @*/
}
```

Note that the translated JML pattern does not work when the equality test is based on value equality between objects. This is the case when the type of the field is `String` where the equality test should be based on the method `equals` rather than ==. One way to have another version of the JML pattern that

uses `equals`, so that the mapping chooses the right pattern based on the type of the property.

## 5 Conclusion

We proposed an approach to translating OCL constraints to JML assertions based on the concept of constraint pattern. The main component of our approach is a set of JML constraint patterns implementing OCL constraint patterns. The possible benefits of this approach is enhancing the quality of the translated assertions by expressing them in a more compact way, and support automating the translation. With appropriate tool support, the JML patterns can be used stand alone to facilitate and simplify the development of JML assertions. The pattern-based approach will have to be evaluated with some real examples.

## REFERENCES


Ackermann, J. and Turowski, K. (2006). A library of OCL specification patterns for behavioral specification of software components. In *Proceedings of the 18th international conference on Advanced Information Systems Engineering*, CAiSE'06, pages 255–269, Berlin, Heidelberg. Springer-Verlag.

Avila, C., Flores, G., and Cheon, Y. (2008). A library-based approach to translating OCL constraints to JML assertions for runtime checking. In *International Conference on Software Engineering Research and Practice, Las Vegas, Nevada*, page 403408.

Cheon, Y., Cheon, Y., Leavens, G. T., and Leavens, G. T. (2002). A runtime assertion checker for the Java Modeling language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP 02), Las Vegas*, pages 322–328. CSREA Press.

Cheon, Y., Leavens, G. T., Sitaraman, M., and Edwards, S. (2005). Model variables: cleanly supporting abstraction in design by contract. *Software-practice & Experience*, 35(6):583–599.

Cok, D. R. and Kiniry, J. R. (2004). Esc/java2: Uniting esc/java and jml - progress and issues in building and using esc/java2. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*. SpringerVerlag.

Davis, J. P. and Bonnell, R. D. (2007). Propositional logic constraint patterns and their use in UML-based conceptual modeling and analysis. *IEEE Trans. Knowl. Data Eng.*, pages 427–440.

Frankel, D. (2002). *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA.

Hamie, A. (2004). Translating the object constraint language into the java modelling language. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1531–1535. ACM Press.

Jacobs, B. and Poll, E. (2003). Java program verification at nijmegen: Developments and perspective. In *Nijmegen Institute of Computing and Information Sciences*, pages 134–153. Springer.

Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT*, 31(3):1–38.

Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. R. (2003). How the design of jml accommodates both runtime assertion checking and formal verification. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 262–284.

OMG (2006). *Unified Modeling Language Specification 2.0: Infrastructure*. OMG doc. smsc/06-02-06.

Wahler, M., Koehler, J., and Brucker, A. D. (2006). Model-driven constraint engineering. In *MoDELS Workshp. on OCL for (Meta-)Models in Multiple Application Domains*, pages 111–125.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Reading, MA.