2-1-2008

A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking

Carmen Avila
University of Texas at El Paso, ceavila3@miners.utep.edu

Guillermo Flores, Jr.
University of Texas at El Paso, gflores3@miners.utep.edu

Yoonsik Cheon
University of Texas at El Paso, ycheon@utep.edu

A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking

Carmen Avila, Guillermo Flores, Jr., and Yoonsik Cheon

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking

Carmen Avila, Guillermo Flores, Jr., and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, TX, USA
{ceavila3, gflores3}@miners.utep.edu, ycheon@utep.edu

Abstract – *OCL is a formal notation to specify constraints on UML models that cannot otherwise be expressed by diagrammatic notations such as class diagrams. Using OCL one can document detailed design decisions and choices along with the behavior, e.g., class invariants and method pre and postconditions. However, OCL constraints cannot be directly executed and checked at runtime by an implementation, thus constraint violations may not be detected or noticed, causing many potential development and maintenance problems. In this paper we propose an approach to checking OCL constraints at runtime by translating them to executable JML assertions. The key components of our approach are a set of JML library classes, use of model variables, and a separation of JML assertions from source code. The library classes implement OCL collection types and facilitate a direct mapping from OCL constraints to JML assertions by using model variables. The translated JML assertions are stored in specification files, separate from source code files, to ease change management of OCL constraints and Java source code. Our approach also facilitates a seamless transition from OCL-based designs to Java implementations.*

Keywords: class invariant, pre and postconditions, assertion, runtime assertion checking, object-oriented class library, OCL, JML

1 Introduction

A UML diagram such as a class diagram cannot express a rich semantics of an application being modeled [1]. There is a need for describing additional constraints on the objects and entities present in the model. The Object Constraint Language (OCL) is a textual, formal specification language for specifying the semantics of UML models [13]; OCL specifications are commonly referred to as *constraints*. Using OCL, for example, one can specify the behavior of a class by writing, among other things, class invariants and method pre and postconditions.

As a design notation, however, OCL is not executable, and OCL constraints are not reified to implementation artifacts. This may lead to many problems in development and maintenance, such as inconsistency. For example, as design constraints are not explicitly expressed in source code, a change to source code that causes a drift or deviation from the initial design may not be detected or noticed by the developer.

In this paper we advocate runtime assertion checking as a partial solution to the problem of design drift. We propose to reify OCL constraints to source code in a form that can be executed and checked at run-time. Specifically we translate OCL constraints to executable assertions written in JML. JML is a formal behavioral interface specification language for Java [9], and a significant subset of it can be checked at runtime [1] [4] (see Section 2.2). Assertions translated from OCL constraints can detect violations of design constraints, thus design drifts at run-time. They also provide excellent API documents that are precise and kept synchronized with the implementation. In addition, as evidenced by a recent introduction of the assert statement to the Java language, assertions are recognized as a practical programming tool and are said to be most effective when they are generated from formal specifications such as OCL constraints.

For the translation we use a set of immutable library classes that implement the collection types defined in the OCL standard library [13]. The use of library classes makes the translation intuitive and traceable, as most OCL constraints are directly mapped to the corresponding JML assertions. We also expect the use of library classes facilitate automation of the translation. Another feature of our approach is the way we organize translated assertions. Instead of embedding them directly to source code, we store them in separate specification files; the JML compiler does an appropriate weaving by combining the JML specification files with Java source code files [4]. This organization facilitates change management of both OCL constraints and Java source code; e.g., changes to OCL constraints can be automatically propagated to JML assertions by retranslating or regenerating the JML specification files, and thus having a minimal impact on the implementation, i.e., Java source code. Our approach is facilitated by several language and tool features of JML, in particular, specification-only variables called *model variables* [5] and specification refinement (see Section 4).

There is a few previous work done on runtime assurance of OCL constraints by translating them to programming languages [1] [14]. As in our approach, the common theme here was to define a set of OCL library classes for the translation. However, OCL constraints are typically translated into source code, i.e., a sequence of program statements, not into assertions or annotations; therefore, they cannot be used as source code-level documents, e.g., precise API specifications.

An assertion is a predicate placed in a program to indicate the truth of the assertion at that place [8] [12]. It is used to specify and reason about the correctness of a program both statically, as in Hoare-style pre and post-conditions [8], and dynamically, as in Design by Contract [11] and assert macros or statements [12]. Surprisingly, however, there is not much work done for translating OCL to executable (JML) assertions. One exception is the work of Hamie, which inspired our own work. Hamie suggested translating OCL constraints to JML assertions by defining a mapping for OCL operators [7]. The operators of OCL collections types are mapped directly or indirectly to the methods of JML's collection model types that implement various kinds of container abstractions, such as sets, bags, and sequences. However, the organization, structures, and vocabulary of JML collection types are somewhat different from those of OCL, and it is unclear how this mapping is to be reified into implementation artifacts, e.g., JML assertions directly referring to program variables. The use of both model variables and immutable collection classes proposed in our approach will greatly simplify the implementation of the mapping and also result in a clear and intuitive mapping. For the development of an automated translation tool, we plan to adapt and refine his mapping.

The remainder of this paper is organized as follows. In Section 2 we briefly review OCL and JML through a small example that we will use throughout this paper. In Section 3 we explain the problem of translating OCL constraints to assertions by referring to program states or variables. In Section 4 we describe our approach by applying it to our running example. We also discuss how our approach solves the problems described in Section 3. In Section 5 we mention our on-going evaluation effort, which is followed by a concluding remark in Section 6.

2 Background

2.1 OCL

The Object Constraint Language (OCL) [13] is a text-based, formal specification language extension to UML for specifying constraints or behaviors of UML models that cannot otherwise be expressed by diagrammatic notation. It supplements UML by providing concise and precise expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. As an example, consider the class diagram

shown in Figure 1 that depicts different types of banking transactions along with associated accounts.
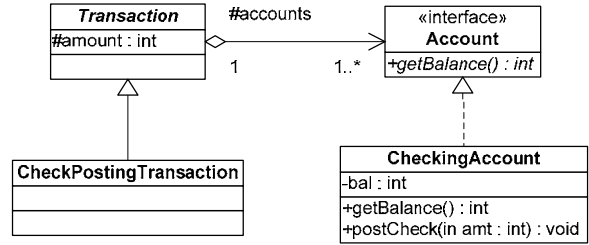

Figure 1. Sample UML class diagram

Different types of transactions and accounts are organized into two class hierarchies, rooted by an abstract class and an interface, Transaction and Account, respectively. The diagram also shows two concrete classes. The class CheckPostingTransaction models a transaction that posts a check to an account. Suppose that a check can be posted to only one checking account and that the account should have enough balance to cover the check to be posted. Then, this constraint can be precisely documented in OCL by writing following statements.

```
context CheckPostingTransaction inv:
    self.accounts->size() = 1 and
    self.accounts->forAll(c: Account | c.isKindOf(CheckingAccount))

context CheckPostingTransaction inv:
    self.accounts->forAll(c: Account | c.getBalance() >= self.amount)
```

As shown, each OCL constraint is preceded by a context specification that identifies the UML model being constrained, in this case CheckPostingTransaction. As indicated by the keyword **inv**, both statements specify class invariants; the first constraint statement states that a CheckPostingTransaction object should be associated with only one CheckingAccount object, and the second statement states that the associated account should have enough balance to cover the check being posted. Note that due to its multiplicity, the account aggregation is viewed as a collection, and thus we can use collection operations such as size and forAll (see below).

OCL comes with several primitives types such as Integer, Real, Boolean, and String and collection types such as Collection, Set, OrderedSet, Bag, and Sequence [13]. In the example constraints above, we used collection operations such as size and forAll; the size operation returns the number of elements contained in a collection, and the forAll operation tests whether an expression is true for all objects of a given collection.

2.2 JML

The Java Modeling Language (JML) is an interface specification language for Java to formally document the

behavior of Java program modules such as classes and interfaces [9]. JML specifications or assertions can be added directly to source code as a special kind of comments called *annotation comments*, or they can reside in separate specification files. In JML, the behavior of a Java class is specified by writing, among others, class invariants and pre and postconditions for the methods exported by the class. The assertions in class invariants and method pre and postconditions are usually written in a form that can be compiled, so that their violations can be detected at runtime [1] [4].

```
// File: CheckingAccount.jml
public class CheckingAccount {
    spec_public private int bal;
    public invariant bal >= 0;

    requires amt > 0 && amt <= bal;
    assignable bal;
    ensures bal == \old(bal) + amt;
    public void postCheck(int amt);

    // the rest of definition
}
```

Figure 2. Sample JML specification

Figure 2 shows a sample JML specification written in a separate specification (.jml) file. It describes the behavior of class SavingAccount. The JML keyword **spec_public** states that the private field bal is treated as public for specification purpose; e.g., it can be used in the specifications of public methods such as postCheck. As shown in the example, 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 most commonly used in the specification of a mutation method such as postCheck that changes the state of an object.

JML supports several features that make it an ideal language to explore our idea of checking OCL constraint at runtime by translating them to executable assertions. As a Design by Contract (DBC) [11] language for Java, it supports class invariants and method pre and postconditions as built-in language features. It combines the practicality of DBC language with the expressiveness and formality of model-oriented specification languages; its powerful assertion language such as various forms of quantifiers will allow us to translate any OCL constraint into a JML assertion. In addition, the vocabulary for writing assertions can be tuned and enriched by add specification-purpose library classes; this is supported by the **model import** clause (refer to Section 4.1 for an example).

JML allows one to write assertions in terms of abstract values provided by model variables [5] (see Section 4.1). There are at least two advantages to writing specifications with abstract values instead of directly using Java variables and data structures. The first is that by using abstract values the specification does not have to be changed when the particular data structure used in the program is changed. Second, it allows the specification to be written even when there are no implementation data structures available.

As shown in the example above, JML assertions can be written in a separate specification file. This not only facilitates the propagation of changes from OCL constraints to automatically-generated JML assertions but also allows one to check OCL constraints even if no Java source code files are available. This also has a practical value because one can ship the object code for a class library to customers, sending the JML specifications but not the source code. Customers would then have documentation that is precise, unambiguous, but overly specific. Customers would not have the code, protecting proprietary rights. In addition, customers would not rely on details of the implementation of the library that they might otherwise glean from the code, easing the process of improving the code in future releases.

JML support the notion of specification refinement for associating multiple specification files to the same source code file or bytecode file (see Section 4). This will allow us to easily add and maintain automatically-generated assertions (from OCL constraints) and manually-written assertions for the same class.

3 The Problem

We translate OCL constraints to executable JML assertions to recognize inconsistencies between a UML design model and its implementation during the development phase and also to detect design drifts during the maintenance phase. The big question then is to translate an OCL constraint to a corresponding JML assertion. As assertions are generally written in terms of program states, we first need to find an appropriate mapping from OCL modeling elements, e.g., the accounts aggregation in the Transaction class, to their representations, e.g., program states or variables, in the implementation classes. As an example, let us consider the CheckPostingTransaction class and its OCL constraints from Section 2.1, and translate them to a Java implementation annotated with JML assertions. Figure 3 shows one such an implementation where JML assertions are directly embedded into the source code as annotation comments (i.e., //@ and /*@ … @*/). The accounts aggregation of its superclass, Transaction, is reified into a JDK set (java.util.Set) with its multiplicity expressed as a class invariant. As easily guessed, our example OCL constraints are also translated into JML invariants. Note that except for a small notational difference and the use of a universal quantifier (**forall**) in place of OCL's forAll operation, the JML assertions are direct translations of the OCL constraints reflecting their structures.

3

```
// File: Transaction.java
import java.util.Set;
public abstract class Transaction {
    /*@ spec_public @*/ protected Set<Account> accounts;
    //@ public invariant accounts.size() > 0;

    // the rest of definition
}

// File: CheckPostingTransaction.java
public class CheckPostingTransaction extends Transaction {
    /*@ public invariant accounts.size() == 1 &&
      @ (\forall a: Account; accounts.contains(a);
      @    a instanceof CheckingAccount); @*/

    /*@ public invariant (\forall a:  Account; accounts.contains(a);
      @    a.getBalance() >= amount); @*/

    // the rest of definitions
}
```

Figure 3. Java implementation with JML annotations

What is wrong with the above translation and resulting assertions? Although at first it looks fine for this particular example, there are several potential problems with such a translation using program variables (i.e., the accounts field) in assertions and adding annotations directly to source code. The main problem is that it may not be always possible to find an appropriate, direct mapping between OCL models and Java representations; e.g., what if the accounts aggregation is implemented as an array? There may be no corresponding Java vocabulary for the OCL terms used in the constraints. In general, OCL constraints have to be recast into the vocabulary defined by a particular choice of representations, e.g., sets, lists, or arrays. The translation is not only hard but also results in assertions with structures different from those of the OCL constraints. Such assertions tend to be lengthy, hard to read and understand, and difficult to be traced back to the original OCL constraints. The translation itself is less amenable to automation.

Worst of all, the translation doesn't accommodate evolution or maintenance of both OCL constraints and Java programs. For example, what happens if the representation becomes changed, e.g., from sets to arrays? The whole JML assertions might have to be rewritten in terms of the vocabulary given by the new representation, i.e., arrays. Similarly, it is also difficult to propagate changes of OCL constraints to the corresponding JML assertions embedded in the source code. Embedding JML assertions directly into the source code also aggravates the problem because it hinders automated tool support for change propagations in both directions.

4   Our Approach

The key idea of our approach is to introduce a new JML library that implements the standard OCL library such as collection types and to store the translated assertions in JML specification files, separately from Java source code

files (see Figure 4). The introduction of OCL-like library classes to JML enables us to map OCL constraints to JML assertions in a one-to-one fashion by preserving the original structures and using almost the same vocabulary. The specific technique is to write JML assertions not in terms of Java program states, i.e., program variables, but in terms of their abstractions using the library classes. In JML, this abstraction is called a *model variable* [5]. A model variable is different from a Java program variable in two aspects. First, it is a specification-only variable meaning that it can be referred to only in assertions, but not in program code. Second, its value is not directly manipulated using assignment statements but is given implicitly as a mapping from a program state, called an *abstraction function* (see Section 4.1 below for an example). In summary, for a UML modeling element such as an aggregation, we introduce a JML model variable of an appropriate type and translate OCL constraints written in terms of the UML element into JML assertions written in terms of the corresponding model variable. In the following subsection, we will illustrate our approach in detail by using our running example.
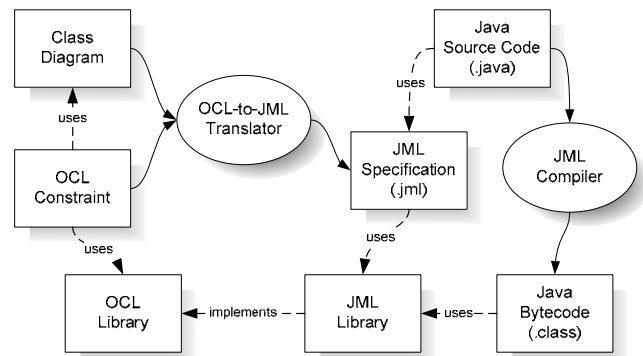

Figure 4. Approach to translating OCL into JML[1]

4.1   Illustration

Let us apply our approach to the transaction classes that we have been playing with. Remember that the abstract class Transaction has an aggregation named accounts, representing the set of accounts involved in a transaction (see Figure 1), and both of the OCL constraints are written in terms of this aggregation. As shown in Figure 5 below, we introduce a JML model variable for this aggregation. The model variable has the same name as that of the aggregation and is of type OclSet. The OclSet class is from our new JML library and implements OCL's set. The rest of the specifications are identical to the previous one except for renaming of the method to follow the OCL's naming convention.

---

[1] We expect that a significant portion of the translation can be automated, and we have a plan for developing such an automated translation tool.

```
// File: Transaction.jml
model import ocljml.OclSet;
public abstract class Transaction {
    spec_public protected int amount;
    public model OclSet<Account> accounts;
    public invariant accounts.size() > 0;
}

// File: CheckPostingTransaction.jml
public class CheckPostingTransaction extends Transaction {
    public invariant accounts.size() == 1
        (\forall a: Account; accounts.includes(a);
         a instanceof CheckingAccount);
    public invariant (\forall a:  Account; accounts.includes(a);
        a.getBalance() >= amount);
}
```

Figure 5. JML specifications from OCL constraints

How is the value of a model variable such as accounts defined? In other words, how can the assertions written in terms of a model variable can be checked at runtime? For a model variable to be executable, it should be provided with a so-called *abstraction function* that specifies its value as a mapping from a program state, i.e., program variables [5]. For example, the abstraction function for the model variable accounts can be specified in the source code of class Transaction as follows.

```
// File: Transaction.java
//@ refines "Transaction.jml";
public abstract class Transaction {
    protected int amount;
    private Account[] accountsRep; //@ in accounts;
    //@ private represents accounts <- OclSet.convertFrom(accountsRep);
}
```

The **refines** statement states that this file refines the given JML specification file, thus inheriting all its assertions such as class invariants and method specifications. The abstract function is specified using the **represents** clause. It maps the array representation (accountsRep) to a set abstraction (accounts). The static method convertFrom creates an OclSet object from an array. The **in** clause specifies a so-called *data group* [10] and states that any method that can modify the model variable accounts can also modify the program variable accountsRep. In addition to the abstraction function, additional implementation invariants (e.g., no duplicates) can also be specified in terms of the representation variables.

How does our approach solve the problems associated with translating OCL constraints into JML by referring to program variables? Note that even if the accounts aggregation is represented as an array, in JML assertions it is still viewed and manipulated as a set as in OCL constraints. Our approach thus clearly alleviates the problems of readability, understandability, traceability, and translation automation, as OCL constraints are one-to-one mapped to JML assertions preserving the structures and also using almost the same vocabulary. Let's next consider

the problem of evolution and maintenance. Let's first consider a change to the implementation, say the representation from an array to a tree. This change is localized, as all we need to do is to rewrite the **represents** clause to define a new abstraction function for the tree. The rest of the specification, in particular, assertions translated from OCL constraints remain the same, as they were written in terms of the model variables. How about changes to OCL constraints? They also have a minimal impact and are localized in that we only need to rewrite the corresponding assertions in the specification files or, with automated translation, regenerate the whole specification files; i.e., there is no or little need to change the source code files.

4.2   JML Library for OCL Collection Types

We implemented in Java all collection types defined in the OCL standard library. Our classes are organized into a class hierarchy with an abstract class OclCollection at the root; other collection classes include OclSet, OclOrderedSet, OclBag, and OclSequence. Since our intention is to use them as JML model classes, all of them immutable; i.e., there is no method that can change the values of these classes. For each collection class, we implemented all the operations defined by OCL except for operations such as forAll (see below). In addition, we defined a set of conversion methods such as convertFrom to convert Java arrays and collections to our implementation of OCL collection types.

In OCL, there are a number of collection operations called *iterators* that take OCL expressions as parameters and work on all elements of a collection. Operations such as select, reject, collect, forAll, and exists fall in this category. Because Java doesn't support this kind of (higher-order) methods, no such methods are defined in our implementation. Instead, they are translated indirectly into JML expressions; e.g., operations such as forAll and exists are translated into JML quantifiers as done in our example.

5   Evaluation

Our implementation of OCL library classes as described in Section 4.2 has several limitations and notable features. First, as the current version of JML doesn't support generics introduced in Java 1.5 [2] (refer to the JML website at http://www.jmlspecs.org), all the collection classes are implemented as so-called raw types. This works well for all the classes and methods except for the sum method of the Collection type. The sum method returns the sum of all the elements contained in the collection. The OCL standard states that each element of the collection must be of a type supporting the binary addition (+) operation and the return type must be the element type given as a type parameter [13]. This causes a trouble in our raw type implementation, OclCollection, as no type parameter is available denoting the element type. We can't specify the exact return type and we can't make any

assumptions about the elements. Our solution is to specify the most general type, i.e., Object, as the return type and check each element's runtime type for the addition compatibility. Depending on the types of elements, the sum is returned as either a Long or Double object; if at least one element is not addition-compatible, then an IllegalStateException is thrown.

Second, as mentioned in Section 4.2, OCL defines a set of iterator operations such as select, reject, collect, forAll, and exists that take an OCL expression as a parameter. Because Java doesn't yet support this kind of (higher-order) methods, no such methods are defined in our implementation. We believe that this problem can be solved when Java 1.7 supports a form of closure called a *code block* [6]. We also proposed to the JML developers to introduce a limited form of OCL-like iterators such as select, collect, and reject which, if adopted, will make the translations of these iterators more direct and natural.

Third, some of OCL collection types such as Set and Sequence define an equals method, and the method is overloaded in that it takes an argument of the same type. In our implementation, however, we followed the Java convention and overloaded the equals method; i.e., its argument type is the class Object, thus overriding the one inherited from the Object class.

Last, in addition to the methods defined in OCL, our implementation adds several new methods such as convertFrom to enable conversion from Java arrays and collections to OCL collections (see Section 4.1).

We noticed several deficiencies in OCL specifications of some of collection operations. For example, operations such as first and last of types Sequence and OrderedSet are partial in that they are defined only when the sequence or ordered set is not empty. However, a precondition asserting this fact, e.g., self->nonEmpty(), is missing from the standard [13]. The append, preprend, insertAt, and subOrderedSet of type OrderedSet also have missing preconditions, and the at method of types Sequence and OrderedSet have missing postconditions.

We are currently evaluating our approach through case studies. Our plan is to perform both quantitative and qualitative measurements to evaluate the effectiveness and efficiency of our approach. In particular, we are interested in knowing the percentage of OCL constraints that we can translate with our approach and the quality of the translated JML assertions. We will also measure the runtime efficiency of the translated assertions that use our new JML library classes. The secondary goal of our evaluation is to gain more insights on our approach, especially its support for and limitations on automation, prior to a full-blown development of an automated OCL-to-JML translation tool.

6   Conclusion

We proposed an approach to translating OCL constraints to JML assertions so that violations of design constraints can be detected at runtime. The key components of our approach are (1) new JML library classes implementing OCL collection types, (2) specification-only variables, called model variables, and (3) separation of specifications from source code. Although we are still evaluating our approach, we believe that our library-based approach will enhance the quality of the translated assertions, accommodate constraint and implementation changes rather than avoiding them, and support translation. Our approach will assist in coping with the plaguing problem of design-implementation inconsistencies, through runtime assurance.

Acknowledgement

References

[1]   D. Arnold, "*C#/OCL Compiler Website*," available from at http://www.ewebsimplex.net/csocl, February 23, 2007.

[2]   G. Bracha, "*Generics in the Java Programming Language*", February 2004, last retrieved on May 14, 2008 from http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[3]   L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, 7(3):212-232, June 2005.

[4]   Y. Cheon and G. T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)," in *Proceedings of International Conference on Software Engineering Research and Practice*, pp. 322-328, Las Vegas, Nevada, June 2002.

[5]   Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software-Practice & Experience*, 35(6):583-599, May 2005.

[6]   D. Flanagan, "*The Open Road: Looking Ahead to Java 7*", August 2007, http://today.java.net/pub/a/today/2007/08/09/looking-ahead-to-java-7.html, retrieved on May 14, 2008.

[7]   A. Hamie, "Translating the Object Constraint Language into the Java Modeling Language," in *Proceedings of the ACM Symposium on Applied Computing*, pages 1531-1535, 2004.

[8]   C.A.R. Hoare, "An axiomatic basis of computer programming," *Communications of ACM*, 12(10):576-580, October 1969.

[9]   G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, 31(3):1-38, March 2006.

[10] K. R. M. Leino, "Data groups: specifying the modification of extended state," *OOPLA '98 Conference Proceedings, SIGPLAN Notices*, 33(10):144-153, October, 1998.

[11] B. Meyer, "Applying design by contract," *Computer*, 25(10):40-51, October 1992.

[12] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, 21(1):19-31, January 1995.

[13] J. Warmer and A. Kleppe, "*The Object Constraint Language: Getting Your Models Ready for MDA*," 2nd edition, Addison-Wesley, 2003.

[14] J. Warmer and A. Kleppe, "*Octopus Website: OCL Tool for Precise UML Specifications*," http://www.klasse.nl/octopus, last retrieved on February 23, 2007.