Checking Design Constraints at Run-time Using OCL and AspectJ

Yoonsik Cheon, Carmen Avila, Steve Roach, and Cuauhtemoc Munoz

TR #09-35
December 2009

Keywords: design constraints, runtime checking, class invariants, pre and postconditions, aspect-oriented programming, Object Constraint Language (OCL), AspectJ language.

1998 CR Categories: D.2.4 [*Software Engineering*] Design Tools and Techniques—modules and interfaces, object-oriented design methods; D.2.4 [*Software Engineering*] Software/Program Verification—assertion checkers, class invariants, formal methods; programming by contract; D.3.2 [*Programming Languages*] Language Classifications—object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—assertions, invariants, pre- and post-conditions, specification techniques.

This is an extended version of a paper appeared in *ITNG 2009: 6th International Conference on Information Technology, April 27-29, 2009, Las Vegas, NV*, pages 223-228. To appear in a special issue of the *International Journal of Software Engineering*, 2(3):5-28, December 2009.

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

Checking Design Constraints at Run-time Using OCL and AspectJ

Yoonsik Cheon[(1)], Carmen Avila[(1)], Steve Roach[(1)], and Cuauhtemoc Munoz[(1)]

(1) Department of Computer Science, University of Texas at El Paso, El Paso, Texas, U.S.A.
E-mail: {ycheon, sroach}@utep.edu, {ceavila3, cuauhtemocm}@miners.utep.edu

ABSTRACT

Design decisions and constraints of a software system can be specified precisely using a formal notation such as the Object Constraint Language (OCL). However, they are not executable, and assuring the conformance of an implementation to its design is hard. The inability of expressing design constraints in an implementation and checking them at runtime invites, among others, the problem of design drift and corrosion. We propose runtime checks as a solution to mitigate this problem. The key idea of our approach is to translate design constraints written in a formal notation such as OCL into aspects that, when applied to a particular implementation, check the constraints at run-time. Our approach enables runtime verification of design-implementation conformance and detects design corrosion. The approach is modular and plug-and-playable; the constraint checking logic is completely separated from the implementation modules which are oblivious of the former. We believe that a significant portion of constraints translation can be automated.

Keywords: AspectJ, Class invariants, Object Constraint Language, Pre and postconditions, Runtime checking

1- INTRODUCTION

One of the problems associated with software development and maintenance is *design corrosion* [28] or *design decay* [13]. In general, the corrosion of software design is proportional to the development and maintenance time when an initial design of software gets modified to accommodate new or changed requirements or to correct defects. Frequently, modifications are implemented by developers other than the original designer, and these developers might not have complete understanding of the original design. Design corrosion also occurs as the result of code hacks and workarounds, a common practice of software maintenance. The real problem is that design corrosion or drift may occur without being noticed by software developers or maintainers. In short, even with rigorous development and maintenance, software often loses its original design and becomes difficult to understand and modify.

The Object Constraint Language (OCL) [27] [33] is a textual notation to specify constraints or rules that apply to UML models such as class diagrams [31]. It is based on mathematical set theory and predicate logic and can express relevant information about the systems being modeled that cannot otherwise be expressed by diagrammatic notations. Using a combination of UML and OCL, one can build a precise design model that includes detailed design decisions and choices along with the semantics such as class invariants and operation pre- and postconditions. Such a precise model is the key to a model-driven development approach [5] [25], the essence of which is to use a model as the basis of software development.

As a design notation, however, OCL is not executable, and OCL constraints are not reified to implementation artifacts. This may lead to problems such as inconsistency during development and maintenance. For example, if design constraints are not explicitly expressed in source code, source code modifications may allow deviations from the design constraints due to a developer oversight or misinterpretation. In this paper we advocate runtime checking as a partial solution to the problem of design corrosion.

We propose to reify OCL constraints to implementations in a form that can be executed to detect violations of design constraints, thus design corrosion, at run-time. As evidenced by the presence of the *assert* statement in the Java programming language, runtime assertion checking is recognized as a practical programming tool and is most effective when assertions are generated from formal specifications such as OCL constraints. We also hypothesize that, with a suitable framework in place, a wide class of important design constraints and properties written in OCL can be automatically translated to runtime checking code.

However, for runtime constraint checking to be effective and practical, it must satisfy several requirements including *transparency*, *modularity*, and *plug-and-playability*. Transparency is essential for any kind of runtime checks. The execution of checking code should be transparent in that, unless a constraint is violated and except for performance measures such as time and space, the behavior of the program should not be changed; that is, the checking code should be free from side effects. The other two requirements are practical considerations. Modularity implies that constraint checking code is organized into modules separated from the program to improve maintainability by enforcing boundaries between the checking code and the program. This eliminates in-line assertions such as *assert* statements as a viable implementation approach. Constraint checking should be plug-and-playable so that the checking code can be added or removed from an implementation without modifying the source code. This will enable the checking code to be applied to different implementations of the same design and also selectively enabled or disabled, for example, for production code.

Our approach is aspect-based in that we translate OCL constraints to AspectJ aspects, which exist separately from the design implementation. AspectJ [21] is an aspect-oriented extension of the Java programming language (see Section 2-2). We call such an aspect a *constraint checking aspect*. The beauty of our approach is that the implementation is independent of the constraint checking aspects. However, when compiled with the aspects, it will be checked for OCL constraints at appropriate execution points at run-time. This leads to more cohesive and readable implementations. In summary, our approach is modular and plug-and-playable. These are the main benefits of using AspectJ as an instrumentation language and differentiate our approach from assertion-based approaches. Another contribution of our work is that we define translation rules from OCL to AspectJ and identify several issues in the use of AspectJ as an instrumentation language.

The remainder of this paper is structured as follows. In Section 2 we give background information on UML, OCL, and AspectJ needed to read the rest of this paper. We also introduce a simple program consisting of three classes to be used as a running example. In Section 3 we describe our approach by first explaining the organization of constraint checking code and then illustrating translations of key OCL constructs and expressions to AspectJ. In Section 4 we describe a case study used to evaluate our approach. In Section 5, we discuss several interesting issues and problems that we encountered during our study. We conclude our paper with related work in Section 6.

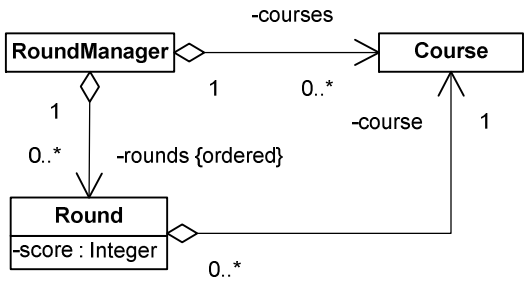

Figure 1 A round manager application

2- BACKGROUND

2-1 UML and OCL

The Unified Modeling Language (UML) [31] is a standard modeling language for writing a software system's blueprints, including its structure and behavior. It uses diagrammatic notations to express various design aspects of a software system; UML 2.0 has 13 different types of diagrams. The diagram shown in Figure 1, for example, is a UML class diagram for an application that keeps track of golf rounds played by a golfer. A class diagram describes the static structure of a system in terms of its components and their relation-

ships. This diagram shows three classes (RoundManager, Course, and Round) and relationships among them. A round manager is composed of a set of courses and an ordered collection of rounds, and each round is associated with a course.

A UML diagram alone, however, cannot express a rich semantics of and all relevant information about an application. The class diagram above, for example, doesn't express the fact that each round was played on a golf course known to the round manager. It is very likely that a system built based only on the diagrams will be incorrect. Thus, there is a need for describing additional constraints on the objects and entities present in the model.

OCL [33] is a textual, declarative notation to specify constraints or rules that apply to UML models such as class diagrams. OCL is based on mathematical set theory and predicate logic. The fact that each round should have been played on a known course can be expressed in OCL as follows.

```
context RoundManager
  inv: rounds->forAll(r: Round | self.courses.includes(r.course))
```

As shown, an OCL constraint is preceded by a context specification that identifies the UML model being constrained, in this case RoundManager. As indicated by the keyword inv, this constraint, called an *invariant*, states a fact that should be always true in the model. The keyword *self* denotes the object being constrained by an OCL expression, called a *contextual instance*; in this case it is an instance of the RoundManager class. Note that due to their multiplicities, the rounds and courses aggregations are viewed as collections, and thus one can use collection operations such as forAll and includes. OCL comes with several primitive types such as Integer, Real, Boolean, and String and collection types such as Collection, Set, OrderedSet, and Sequence (see Section 4). The example uses collections opertations such as *forAll* and *includes*; the *forAll* operation tests whether an expression is true for all objects of a given collection, and the *includes* operation tests whether an object is contained in a collection.

It is also possible to specify the behavior of an operation in OCL. For example, the following OCL constraints specifies the behavior of an operation named addRound by writing a pair of predicates called *pre* and *postconditions*.

```
context RoundManager::addRound(r: Round): void
  pre: courses->includes(r.course)
  post: rounds = rounds@pre->append(r)
```

The pre and postconditions pair states that, given a round played on a known course, the operation should append the round to the list of known rounds. The postfix operator @pre denotes the value of a property (rounds) in the pre-state, i.e., just before an operation invocation. The OCL *append* operation adds an object to the end of an ordered collection.

2-2 AspectJ

AspectJ [1] [21] is an aspect-oriented extension for the Java programming language to address crosscutting concerns. A *crosscutting concern* is a system-level, peripheral requirement that must be implemented by multiple program modules, thereby leading to tangled and scattered code. Examples of crosscutting concerns include logging, security, persistence, and concurrency. AspectJ provides built-in language constructs for implementing crosscutting concerns in a modular way. The key idea is to denote a set of execution points, called *join points*, and introduce an additional behavior, called an *advice*, at the join points. The following code shows an AspectJ aspect that checks the precondition of the addRound method described earlier.

```
public aspect PreconditionChecker {
  pointcut addRoundExe(RoundManager m, Round r):
    execution(void RoundManager.addRound(Round)) && this(m) &&
args(r);

  before(RoundManager m, Round r): addRoundExe(m, r) {
    if (!m.hasCourse(r.getCourse()))
      throw new RuntimeException("precondition error");
  }
}
```

The pointcut declaration designates a set of execution points and optionally exposes certain values at those execution points. The pointcut addRoundExe denotes executions of the addRound method and exposes the receiver (m) and the argument (r). The *before* keyword introduces an advice that is to be executed before the execution of a join point; there are also *after* and *around* advices [21]. In the example, the advice is executed right before the execution of the addRound method and checks its precondition by referring to the values exposed at that join point. If the RoundManager class is compiled with the above aspect, all invocations of the addRound method that violate the precondition will be detected and result in runtime exceptions. One advantage of aspect-oriented programming is that the base code such as the RoundManager class doesn't depend on the aspect code such as the PreconditionChecker aspect; in fact, the former is oblivious of the later in that it doesn't even know the existence of the aspect. In summary, aspect-oriented programming provides a modular solution to crosscutting concerns.

3- APPROACH

In this section we explain our approach to monitoring design constraints by applying it to the round manager application. We first describe how we organize our AspectJ code and then explain how we translate OCL constructs and expressions to AspectJ checking code. Figure 2 shows our framework for checking OCL constraints. For each class we have a separate aspect that advises the class. This aspect, called a *constraint checking aspect*, is re-

sponsible for checking all OCL constraints specified for the class. Each constraint checking aspect is defined to be a subclass of an abstract class, OclChecker. This class provides a set of utility methods, such as a mechanism for reporting constraint violations, to constraint checking aspects. It uses the strategy design pattern [15] to separate constraint checking from violation reporting and to select a reporting mechanism appropriate for a particular application; for example, a constraint violation may be reported by throwing an exception, logging it, or notifying it to another program. OCL constraints are side-effect free, and there is no code dependency between the class being checked and its constraint checking aspect. The class is independent of the aspect, and the same aspect can be applied to different implementations of the same design. Constraint checking can also be easily added or removed by recompiling the source code. These are the main benefits of using AspectJ as an instrumentation language. In the following subsection we show a sample constraint checking aspect.

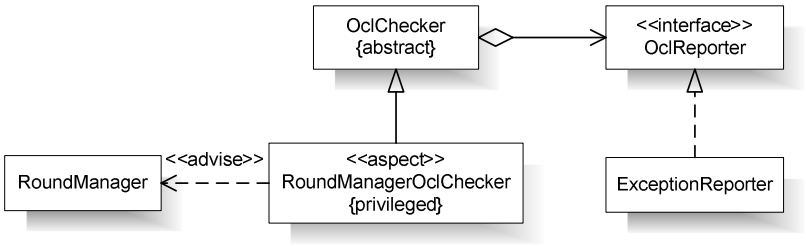

Figure 2 Framework for checking OCL constraints

3-1 Illustration

To illustrate our approach, let us consider the following OCL constraint that specifies the behavior of the course() method of the RoundManager class.

context RoundManager::course(): Set(Course)
  post: **result** = courses

The keyword *result* denotes the return value of the operation. Figure 3 shows the constraint checking aspect for the RoundManager class. It only shows code snippet relevant for the checking of the above OCL constraint. The aspect is declared to be *privileged*, which means that it bypasses Java language access checking and thus can access private members. This allows the aspect, for example, to access private fields such as courses used in the postcondition.

As shown, an OCL constraint is translated to pointcuts and advices, often along with helper fields or methods. The pointcuts define execution points at which constraint checks are to be performed, and the advices perform constraint checks. It is also common that pointcuts expose several values such

as the receiver and arguments at the execution points so that they can be referred to in the constraint checking advices. For example, the pointcut coursesExe exposes the receiver, a RoundManager, on which the courses method is invoked. The advices check OCL constraints and thus are often direct translations of the constraints. In the example, a special form of the *after* advice was used to refer to the return value; this *after-returning* advice gets executed only if the join point returns normally without throwing an exception. A constraint violation, if detected, is reported by calling a framework method such as checkPost; the AspectJ pseudo variable *thisJoinPoint* denotes the join point currently being advised, i.e., an invocation of the course() method. The framework method inherited from the abstract class OclChecker checks the given condition and reports an appropriate constraint violation, e.g., by throwing an exception.

```
public privileged aspect RoundManagerOclChecker extends OclChecker {

  pointcut coursesExe(RoundManager m):
    this(m) && execution(Set<Course> RoundManager.courses());

  after(RoundManager m) returning (Set<Course> s): coursesExe(m) {
    checkPost(s.equals(m.courses), "post: result = courses", thisJoinPoint);
  }
}
```

Figure 3 Constraint checking aspect

In the next two subsections we explain how we translate OCL constructs and expressions to AspectJ pointcuts and advices.

3-2 Translating OCL Constructs

Here we describe our approach for translating various OCL constructs such as class invariants and operation pre and postconditions.

Invariants: An invariant is a boolean expression that states a condition that must always be met by all instances of the type for which it is defined [33]. The question is when to check an invariant. Since an invariant constrains the state of an object and an object's state is represented by fields, an invariant may be checked whenever fields get new values. For example, an OCL invariant "**context** Round **inv**: score > 0," stating that the score must be positive, may be translated to the following AspectJ advice; the keyword *set* denotes an execution point at which a new value is assigned to a field.

```
after(Round r): set(int Round.score) && this(r) {
  checkInv(r.score > 0);
}
```

This approach, however, has several problems. First, it does not work for objects with reference fields. The states of such objects may be changed indirectly by mutating the contained objects, i.e., field objects. This kind of state changes cannot be captured using the *set* pointcut because they occur without assigning new values to the fields of objects under consideration. Second, the approach is too restrictive. It ignores Java's control structures and abstraction mechanisms and does not provide for internal or hidden program states. For example, a temporary violation of an invariant in hidden program states such as during the execution of a method body is not allowed. As a result, a circular data structure cannot be created because its creation requires more than one assignment and the first assignment may lead to an invariant violation. Third, the approach may be inefficient, as every assignment to a field has to be checked for an invariant violation.

To remedy the aforementioned problems, we follow the Design by Contract principle [26] where a class invariant must be established upon the completion of an object construction and be preserved by every method invocation on the object. In other words, an invariant should be established by a constructor of a class and be preserved by every method of the class.

We translate an invariant into a pair of *before* and *after* advices. The *before* advice checks the invariant on the pre-state, i.e., right before the execution of the invoked method, and the *after* advice checks the invariant on the post-state, i.e., right after the execution of the invoked method or constructor. For example, the OCL invariant on the RoundManager class in Section 2-1 stating that each round should be played on a known course can be translated to the following AspectJ code.

```
pointcut constructorExe(RoundManager m):
  this(m) && execution(RoundManager.new(..));

pointcut methodExe(RoundManager m):
  this(m) && execution(* RoundManager+.*(..));

before(RoundManager m): methodExe(m) {
  checkAllInv(m);
}

after(RoundManager m): constructorExe(m) || methodExe(m) {
  checkAllInv(m);
}

private void checkAllInv(RoundManager m) {
  for (Round r: m.rounds)
    checkInv(m.courses.contains(r.getCourse()));
}
```

In the pointcut declarations wild card symbols ("*" and "..") are used to denote executions of any constructor or method, regardless of its name, return type, or arguments; the keyword *new* denotes a constructor. In the pointcut methodExe, the notation RoundManager+ denotes the class RoundManager and all its subclasses. The intention is to make the invariant be preserved by subclasses. That is, the invariant should also be maintained by the additional methods of subclasses—methods that are overridden or introduced by subclasses (see Section 5 for a discussion on constraints inheritance). Note also that the *before* advice is applied only to method executions, but not to constructor executions.

Pre and postconditions: The pre and postconditions specify the behavior of an operation and should be checked right before and after the execution of the operation. They can be translated to a pair of before and after advices (see earlier examples in Sections 2-2 and 3-1) or to a single around advice. An *around* advice is the most powerful kind of advice and surrounds a join point such as a method execution. It can perform custom behavior before and after the join point and is also responsible for choosing whether to proceed to the join point or to shortcut executing it by returning its own return value or throwing an exception. For example, the pre and postconditions of the addRound method of the RoundManager class can be translated to the following around advice (see Section 2-1).

```
pointcut addRoundExe(RoundManager m, Round r):
  this(m) && args(r) && execution(void RoundManager.addRound(Round));

void around(RoundManager m, Round r): addRoundExe(m, r) {
  // pre: course->includes(r.course)
  checkPre(m.courses.contains(r.getCourse()));

  List<Round> oldRounds = new ArrayList<Round>();
  oldRounds.addAll(m.rounds); // rounds@pre
  proceed(m, r);
  oldRounds.add(r);

  // post: rounds = rounds@pre->append(r)
  checkPost(m.rounds.equals(oldRounds));
}
```

Note that the pointcut declaration exposes the receiver and the argument so that they can be referred to in the advice. The advice first checks the precondition, proceeds to continue with the normal flow of execution at the corresponding join point (as indicated by the keyword *proceed*), and finally checks the postcondition. If there is any pre-state value referenced in the postcondition (e.g., self@pre), it is cloned or copied to a local variable in the pre-state (i.e., prior to proceeding) so that they can be used for the postcondition check in the post-state (see Section 5 for a discussion on this).

Initialization: The OCL *init* construct specifies the initial value of an attribute or association end, both of which are usually mapped to Java fields. For example, the following constraint states that the initial value of the attribute *course* of the RoundManager must be an empty set.

```
context RoundManager::courses
  init: Set{}
```

The **init** construct can be translated to an *after* advice on constructor executions, which is the point when an object completes its initialization. For example, the above constraint is translated to the following AspectJ code.

```
pointcut constructorExe(RoundManager m):
  this(m) && execution(RoundManager.new());

after(RoundManager m): constructorExe(m) {
  checkInit(new Set<Round>.equals(m.courses));
}
```

Definition, derivation, and operation body: The *def* construct introduces a new attribute or query operation to a UML model such as a class diagram. It also specifies the value of the new attribute or the return value of the new query operation. For example, the following OCL statement introduces a new query operation named *rounds* to the RoundManager class. The operation takes a course and returns all the rounds played on that course.

```
context RoundManager
  def: rounds(c: Course): Sequence(Round) = rounds->select(r: Round |
     r.course.name = c.name)
```

The OCL collection operation *select* returns a new collection consisting of all the elements from a collection that satisfy a given condition. As the OCL *def* construct constrains the result of a newly-introduced operation, it is translated to an *after-returning* advice that is applied only when the corresponding join point terminates normally without throwing an exception, as shown below.

```
pointcut roundsCourseExe(RoundManager m, Course c):
  this(m) && args(c)
    && execution(List<Round> RoundManager.rounds(Course));

after(RoundManager m, Course c) returning (List<Round> l):
  roundsCourseExe(m, c) {
  List<Round> expected = new ArrayList<Round>();
  for (Round r: m.rounds)
    if (r.getCourse().equals(c))
      expected.add(r);
  checkDef(l.equals(expected));
}
```

In the case of an attribute definition, it states how the value of the newly-introduced attribute must be calculated. For example, we may introduce a new attribute named numOfRounds to explicitly keep track of the number of known rounds, as follows.

```
context RoundManager
  def: numOfRounds: Integer = rounds.size()
```

The translation of an attribute definition depends on how the newly-introduced attribute is implemented in the program. If it is implemented as a query method, it can be translated to an *after-returning* advice that checks the return value. On the other hand, if it is implemented as a field, it can be translated to a check that is executed when the field is read, as follows.

```
after(RoundManager m) returning (int r):
  this(m) && get(int RoundManager.numOfRounds) {
  checkDef(r == m.rounds.size());
}
```

OCL has two more constructs. The *derive* construct specifies the value of a derived attribute or association end. Its translation is similar to that of the attribute definition. The *body* construct defines the result of a query operation, and thus it can be treated like an operation definition.

3-3 Translating OCL Expressions

We believe that, given a mapping between OCL modeling elements and Java implementation artifacts, most OCL expressions can be systematically translated to AspectJ expressions and statements. The operators of OCL built-in types such as Boolean and Integer can be semi-automatically translated to corresponding operators of Java. It should be noted, however, that OCL uses a three-value logic to handle undefinedness [27], and thus care should be taken when translating Boolean expressions. For example, an OCL expression *x* or *y* cannot be directly translated to the corresponding Java expression *x* || *y*. If *x* becomes undefined, the result is *y* for the OCL case, while it is undefined for Java; in Java, for example, if the evaluation of x encounters an exception then the evaluation of the whole expression terminates abruptly by throwing the exception. The OCL expression should be translated to a block of code such as the following, where r contains the result [8]:

```
boolean r = false;
try { r = EVAL(x); } catch (Exception e) {}
if (!r) { r = EVAL(y); }
```

As shown above, collection operations such as *forAll*, *select*, *includes*, and *appends* may also be systematically translated to Aspect code. Since Java doesn't support blocks or closures to specify the conditions, iteration opera-

tions such as forAll and select must be transformed into sequences of statements.

OCL also has an interesting operator that can be used only in the postcondition. The *hasSent* operator, denoted "^",specifies that a certain interaction has taken place during the execution of an operation. The expression **self**.addRound(r) in the second postcondition below becomes true if an addRound message with argument r was sent to *self* during the execution of the operation. It constrains the traces of an operation execution and can be used to prescribe that the operation be implemented in terms of the addRound(Round) operation.

```
context RoundManager::addRound(s: Sequence(Round)): void
  pre: s->forAll(r: Round | self.courses->includes(r.course))
  post: rounds = rounds@pre->union(s)
  post: s->forAll(r: Round | self^addRound(r))
```

It is instructive to show how an OCL expression consisting of message sending can be translated to an AspectJ check. The core of our solution is to introduce another advice to trace the execution of the operation and detect the required message sending. However, it is a bit involved because the two advices—execution tracing and constraint checking—have to communicate with each other and the execution of the operation may be recursive.

The tracing advice for the above example is shown below. In essence, it records each round object that was used as an argument to the addRound(Round) method during an execution of the addRound(List<Round>) method on a field oclMessages.

```
private Set<Round> RoundManager.oclMessages = null;

pointcut addRoundListExe():
  execution(void RoundManager.addRound(List<Round>));

pointcut addRoundNestedExe(RoundManager m, Round r):
  this(m) && args(r) && cflowbelow(addRoundListExe())
  && execution(void RoundManager.addRound(Round));

after(RoundManager m, Round r): addRoundNestedExe(m, r) {
  m.oclRoundMessages.add(r);
}
```

The first statement, known as *static crosscutting*, introduces a new field named oclMessages to the RoundManager class. The pointcut addRoundNestedExe denotes an execution of the addRound(Round) method occurring during an execution of the addRound(List<Round>) method; the keyword *cflowbelow* denotes all the join points enclosed by the argument join point.

The constraint checking advice shown below uses the information accumulated by the tracing advice. For this, it refers to the newly introduced field oclMessages.

```
void around(RoundManager m, List<Round> l):
  this(m) && args(r) && addRoundListExe() {

  Set<Round> oldMessages = m.oclMessages;
  m.oclMessages = new HashSet<Round>();

  proceed(m, l);

  checkPost(m.oclMessages.containsAll(l));
  if (oldMessages != null)
    m.oclMessage.addAll(oldMessages);
  else
    m.oclMessages = null;
}
```

One complication is that because the join point (i.e., execution of the addRound(List<Round>) method) may be executed recursively, we have to save and restore the value of the field oclMessages before and after proceeding to the join point. Upon the completion of the join point execution, we also update the information stored in the field oclMessages; all the messages that were sent during a recursive execution were also sent during the execution that initiated the recursion.

4- CASE STUDY

We performed a case study to evaluate the feasibility and effectiveness of our approach. An initial challenge was to find an open-source Java application that has a formal UML model including class diagrams and OCL constraints. Instead of writing a small sample program by ourselves we decided to use an existing program to eliminate subjectivenss in the experiment and to make the case study more realistic. The OCL standard specification defines several collection types such as Collection, Set, OrderedSet, Bag and Sequence [27]. The OCL collection types are organized into a class hierarchy, and the behavior of each type is formally specified in OCL in the standard specification. We also found a couple of Java implementations of the OCL collection types [1] [11]; they are to facilitate an interpretation or translation of OCL constraints. We decided to use the one from the Dresden OCL Toolkit [11] [32] for our case study (see Section 6 for a discussion of the Dresden OCL Toolkit). This implementation consists of five generic classes: OclCollection<T>, OclSet<T>, OclBag<T>, OclOrderedSet<T>, and OclSequence<T>. All classes are immutable in that no method can change the states of objects. All collection operations specified in the standard are implemented except for iterator operations such as *forAll* that take OCL expressions as parameters and work on each element of a collection. This is because Java doesn't support this kind of higher-order methods.

The objective of our case study was to detect inconsistencies between OCL constraints of the collection types and the Java implementations. For our case study, we first translated OCL constraints on collection types into AspectJ constraint checking code by following the translation rules described in Section 4. This was done manually and resulted in one constraint checking aspect per OCL collection type plus several framework classes for checking constraints and reporting constraint violations (see Figure 2). We next devised a suite of test data for each collection type to run the corresponding Java class after applying the constraint checking aspect. This was again done manually but we used JUnit [3] [20] to organize the test suites and automate test execution. In our JUnit tests, we determined test results based on the occurrence of OCL constraints violations detected by the constraint checking aspects. If a test execution results in a pre-state constraint violation error such as a precondition error, the test data is rejected as invalid because prestate constraints are the client's obligation. On the other hand, if a test execution results in a post-state constraint violation error such as a postcondition error, it is a test failure; such a constraint violation error means an inconsistency between the constraint and the implementation, as post-state constraints are the implementer's obligations. In essence we used OCL constraints as test oracles [7] [10] [29]. Table 1 below summarizes the size of various source code given in terms of the numbers of source code lines including comment lines. The framework code is a small set of reusable Java classes supporting AspectJ constraint checking code (see Section 3).

Table 1 Size of source code

Source Code	No. of Lines
OCL constraints	336
Java base code	1787
AspectJ checking code	720
Framework code	143
JUnit test code	1001

The size of the AspectJ constraint checking code excluding the framework is about twice of that of the Java code being checked. Below we summarize what we learned from our case study (see also Section 5 for discussions on some of the issues motioned below).

4-1 Translation of Constraints

We were able to translate most of the OCL constraints to AspectJ constraint checking code. The translation was almost mechanical, though there were several problems that we encountered. Perhaps, the most commonly used OCL operator is the equality operator, e.g., x = y. How is the OCL equality

operator (=) translated to Java? OCL supports only *value equality* while Java supports an additional notion of equality called *reference* or *identity equality.* There seems to be no universal solution to this problem, but the most general approach would be to translate the OCL equality operator to the Java == operator for primitive types such as int and the *equals* method for object types (see an example below); this assumes that each class define an appropriate notion of equality for itself by overriding the inherited *equals* method. The problem of this approach, however, is that often two different notions of equality are required for the same class. For example, a container class such as IdentityHashMap<K,V> compares its key objects for reference equality, and thus the translation will be incorrect for such a class. The context of each use of the quality operator should determine the translation, which will become a barrier to an automatic translation of constraints.

As mentioned in Section 3.3, OCL uses a three-value logic to handle undefinedness in expressions. Because of this, care should be taken when translating OCL constraints. As an example, consider the following constraint stating that two sequences are equal if both sequences contain the same number of elements and the elements are position-wide equal.

```
context Sequence::equals(s: Sequence(T)): Boolean
post: result = Sequence{1..self->size()}->
  forAll(index: Integer | self->at(index) = s->at(index))
  and self->size() = s->size()
```

The expression Sequence{1..self->size()} denotes a sequence containing all the numbers from 1 to self->size() in that order; OCL uses 1-based indices. A naïve translation of the above constraint may produce the following Java code.

```
for (int index = 1; index < self.size(); index++) {
  checkPost(self.at(index).equals(s.at(index));
}
checkPost(self.size() == s.size());
```

There are two problems with the translated code. First, if the size of *self* is bigger than that of *s*, then the OCL constraint evaluates to false. For a certain index value, the s->at(index) expression will evaluate to an undefined value because no element is defined for the index, thus leading to false for self->at(index) = s->at(index) and the first conjunct. As a result, the whole expression will evaluate to false. In the translated Java code, however, one of the s.at(index) calls will result in an exception such as an index-out-of-bound exception, thus the whole expression will evaluate to an undefined value. Therefore, the translation is incorrect. This problem can be fixed for this particular example by moving the second checkPost statement before the *for* loop statement. A similar problem occurs when the argument (s) is undefined or null; for a similar reason, the OCL constraint evaluates to false while the Java code evaluates to an undefined value by throwing a null-pointer exception.

4-2 Effectiveness of the Approach

Our JUnit tests revealed several errors and deficiencies in both the Java implementation of the OCL collection types and the OCL constraints themselves specified in the standard. For example, the following is the behavior of the Set::union operation specified in the OCL standard.

```
context Set(T)::union(s : Set(T)) : Set(T)
  post: result->forAll(elem | self->includes(elem) or s->includes(elem))
  post: self ->forAll(elem | result->includes(elem))
  post: s ->forAll(elem | result->includes(elem))
```

The result should be the union of the receiver and argument sets. The following is the implementation of the union operation by the Dresden OCL Toolkit.

```
public OclSet<T> union(OclSet<T> aSet) {
   OclSet<T> result;
   result = new OclSet<T>();
   result.addAll(aSet);
   return result;
}
```

Obviously, the above code is incorrect with respect to the standard specification, as it returns a new set that contains only the elements of the argument set. This and similar kinds of errors were detected by the translated AspectJ code as inconsistencies between the OCL constraints and the Java implementations.

Our case study also revealed an unexpected benefit of using AspectJ-based approach. One of the weaknesses of program testing and runtime verification is the difficulty of testing or checking missing features. Our AspectJ-based approach was able to detect this kind of error. For example, when we translated and compiled the OCL constraint for the operation Set::union(bag: Bag(T)): Bag(T), we received a compilation warning message stating that our advice for the constraint has not been applied. This warning was caused because there was no such method in the implementation; the return type of the corresponding Java method was OclSet, not OclBag. It may be possible to write AspectJ aspects to check for a presence of features in an implementation. It would be interesting future work to study the extent of missing features that can be detected using aspect-oriented programming.

We also found several deficiencies in OCL specifications of some of the collection operations. For example, operations such as *first* and *last* of types Sequence and OrderedSet are partial in that they are defined only for non-empty sequences and ordered sets. However, a precondition asserting this fact, e.g., self->nonEmpty(), is missing from the standard [27], as shown below.

context Sequence(T):: first():T
post: **result**= **self**->at(1)

The *append*, *preprend*, *insertAt*, and *subOrderedSet* operations of type OrderedSet also have missing preconditions. Missing or loose preconditions were detected as errors during test executions; e.g., attempting to access the first element of an empty sequence resulted in an exception. While manually translating OCL constraints to AspectJ code, we also noticed missing or loose postconditions for some operations; e.g., the *at* operations of Sequence and OrderedSet have missing postconditions. Our approach cannot detect the problem of weak postconditions, as it is a fundamental problem of an assertion-based approach in that a missing assertion can't be checked and thus detected [6].

5- DISCUSSION

In this section we discuss some of the interesting issues, challenges, and problems that we encountered. Some are related to OCL itself, and others are technical questions that require further investigation.

Inheritance of constraints: The OCL standard specification [27] is silent about the inheritance of constraints. However, for a subclass object to behave like a superclass object [24], it is reasonable to let a subclass inherit constraints of all its superclasses, direct or indirect; e.g., a subclass has to preserve the class invariants of its superclasses. We implement the semantics of conjoining inherited invariants. For this, we use wild cards and patterns in pointcut declarations to include join points of (future) subclasses. The invariant pointcut for class *T*, for example, is execution(* *T*+.*(..)); recall that *T*+ means *T* and all its subtypes. As a result, additional methods of *T*'s subclasses are also checked for *T*'s invariants. The integrity of the extended state of a subclass often depends on that of its inherited state. Thus, it is desirable to check the inherited constraints first. We achieve this by explicitly declaring precedence between constraint checking aspects of a subclass and its superclass. Unlike invariants, there is no widely-accepted semantics for the inheritance of pre and postconditions [17]. It should be noted, however, that our way of translating pre and postconditions produces the effect of conjoining inherited pre and postconditions, respectively[1]; this semantic interpretation is called a *partial exception correctness* [17]. This is because the pre and postcondition checking advice for an overridden method of a superclass is also applied to an overriding method of a subclass.

Partial or total correctness. The OCL standard specification [27] is not clear about the exact semantics of pre and postconditions regarding program termination. There are two choices. First, we can consider the postcondition only

[1] A similar semantics is produced if an operation has multiple constraints and each constraint is translated separately; i.e., pre and postconditions are conjoined, respectively.

if the operation terminates. Alternatively, if an operation is invoked in a state where its precondition holds, we must show that the operation terminates in a state where the postcondition holds. This distinction is called *total* and *partial correctness* [18]; total correctness requires termination. In Java, a method invocation may terminate abruptly by throwing an exception. This is not a normal termination, and the result is undefined. According to partial correctness, therefore, such an invocation should not result in a postcondition violation; however, total correctness demands a postcondition violation. Since OCL does not provide a notation for specifying exceptional behavior, we have adopted the partial correctness semantics.

Invariants revisited: In Java, a class can have two kinds of methods, instance methods and class (a.k.a. static) methods. Since an OCL invariant constrains the instances of a class, class methods should not be checked for invariants; class methods cannot refer to instance variables anyway. This can be achieved by restricting the invariant check pointcut to only instance methods, e.g., **execution**(!**static** * T+.*(**..**)). An invariant should be established when an object completes its initialization. This in general happens when a constructor call returns. But, how about a nested constructor call such as *this* or *super* call? In theory, such a call shouldn't be checked for the establishment of an invariant, as the object is still under construction. It can be done by rewriting the constructor invariant pointcut to: **execution**(T.**new**(**..**)) && !**cflowbelow**(**execution**(T.**new**(**..**)). However, the downside is that reasoning about such a call may not be modular because we cannot rely on the invariant. There is a similar concern for method calls made during a constructor execution. Should the invariant be checked before and after such method calls? Perhaps, they shouldn't be, as the object is still under construction. Then again, reasoning becomes non-modular and may lead to a whole program analysis, as specifications such as invariants and postconditions cannot be used in reasoning. Related problems are helper methods and visibility of methods? Should an invariant be preserved by even so-called helper methods? These are auxiliary methods introduced to assist in implementing public methods. How about visibility of methods? Do all methods including private methods, regardless of their visibility, have to preserve the invariant? Again, the concern is the scope of invariants and the modularity of reasoning.

Side-effect freeness: OCL expressions are not allowed to have side-effects. For this, only query operations are allowed in OCL expressions, and all OCL standard types such as Integer and Collection are value types. Special care should be taken to preserve the side-effect freeness of OCL expressions when translating them to Java expressions or statements. For example, the *append* operation of the OCL Sequence type cannot be directly translated to the seemingly correct *add* method of the Java List type. The former creates a new sequence while the latter mutates the list; OCL sequences are immutable while Java lists are mutable. In general, checking side-effect freeness of an expression requires a whole program analysis.

Advice precedence: When there are multiple advices for the same join point, the order in which the advices are applied affects the outcome. AspectJ defines precedence among such advices, based both on the order they appear in an aspect and the precedence among multiple aspects. However, it is possible for a constraint violation to shadow another violation at the same execution point. For example, an exception thrown by a *before* advice of lower precedence is shadowed by an exception thrown by an *after* advice of higher precedence. Thus, a pre-state constraint violation (e.g., a method precondition violation) may be shadowed by a post-state constraint violation (e.g., a post-state invariant violation), which is undesirable. Therefore, constraint checking advices should be carefully ordered in an aspect.

Avoiding infinite recursion: In OCL a query operation can appear in a constraint such as an invariant. If care is not taken while checking such a constraint, it may lead to an infinite loop; e.g., evaluating the invariant itself may initiate another instance of invariant check (caused by the query method call), which again initiates another invariant check, and so on. This kind of infinite recursion can be avoided by excluding the join points enclosed in the constraint checking aspects from constraint checking pointcuts; i.e., cut !**cflow**(**within**(*OclChecker)) should be conjoined to the constraint checking pointcuts.

6- RELATED WORK

Several different approaches are possible for checking design constraints such as OCL constraints against implementations. The most common approach is to map the constraints to the target language by implementing a constraint checker in that language and making it a part of the implementation (see for example [19]). Constraints may also be mapped to executable assertions if the implementation language provides a facility such as the *assert* macro or statement [1] [16]. Below we discuss previous work known to us that utilized aspect-oriented techniques. An important contribution of our work is that we explicitly defined translation rules from OCL constructs and expressions to AspectJ code, and we also identified several unresolved problems and issues that need to be considered for the translation (see Section 5).

Briand, Dzidek, and Labiche described an approach for automatically instrumenting OCL constraints in Java using AspectJ [4] [12]. They defined templates for translating class invariants and operation pre and postconditions to AspectJ advices. Their approach explicitly addresses abrupt termination of method invocations; class invariants are checked—as such invocations should also leave the object in a consistent state—but postconditions are not. The approach also supports inheritance of constraints; as in our approach class invariants are inherited to subclasses and conjoined. However, their implementation strategy is different. For the inheritance of class invariants, for example, they inject an invariant checking method to the target class using AspectJ's member introduction facility or static crosscutting (see Section 3-3), and the injected method makes a *super* call to invoke the invariant check me-

thod of its superclass. Though not explained in their papers, such a super call should be made using Java's reflection facility because the superclass, if not instrumented, will not have the invariant check method. This leads to unnecessary performance overhead. Worse, the approach doesn't work for Java interfaces because an interface cannot contain any method definitions. Regarding operation pre and postconditions, postconditions are inherited to subclasses but preconditions are not. The approach didn't consider OCL 2.0 features such as message sending (see Section 3-3).

Richters and Gogolla presented an approach for monitoring OCL constraints at run-time [30]. An interesting feature of their approach is that monitoring is done at the model level in terms of modeling elements. For this, they mapped implementation actions such as method calls to modeling actions such as operation invocations and checked the validity of modeling actions using an external tool. AspectJ was used to specify pointcuts for state changes and constraint check points, such as object creation, attribute modification, and association link changes; associations were assumed to be reified to fields. Their approach supported only class invariants and operation pre and postconditions; private methods were not checked for class invariants. The strength of their approach is a clear separation of abstraction levels between implementations and their models; a similar benefit was obtained in an assertion-based approach by using a specification-only variable called a *model variable* [2]. However, its weakness is the cost for converting concrete representation values to abstract modeling values, as well as its reliance on an external, heavyweight tool. In their approach, AspectJ is used only to identify constraint check points and extract values at these points. In our approach, however, AspectJ is also used to actually check OCL constraints, and these checks are performed at the implementation level using concrete values. Thus, there is no need to coerce concrete values to abstract values, which often becomes very expensive [9].

Kiviluomua, Koskinen and Mikkonen presented an aspect-oriented approach for monitoring the execution of a program using UML behavioral profiles and AspectJ [22]. Behavioral profiles consist of class diagrams containing role definitions and behavioral rules given as sequence diagrams. The behavioral profiles also bind roles to the actual program classes, and they are translated to AspectJ aspects. This approach does not use OCL constraints, and it only supports translation of role definitions to AspectJ code.

Froihofer et al. reviewed and evaluated different constraint validation approaches for Java [14]. They discussed handcrafted approaches, code instrumentation using OCL and JML [23], aspect-oriented programming, proxy implementations, CORBA, and EJBs. Each approach has its own advantages and disadvantages; e.g., different approaches have different runtime overheads, ranging from a factor of two to more than one hundred.

Demuth and Wilke presented an OCL verification tool called the Dresden OCL Toolkit, consisting of a parser, an interpreter, and a Java code generator

for OCL [11] [32]. The interpreter can be used to verify OCL constraints by interpreting them on a UML model and its implementation. The Java code generator generates AspectJ code, which can be executed to verify the OCL constraints. One key difference between the Dresden tool and our approach is that the Dresden tool tries to generate as much actual implementation code as possible from the OCL constraints while our approach generates constraints checking code. For example, the Dresden tool translates OCL constraints such as *def*, *derive*, and *body* (see Section 3-2) to AspectJ implementation code, e.g., query methods, rather than to assertion checking code. The tool supports Eclipse and several different UML modeling tools.

7- CONCLUSION

We proposed runtime checks as a solution to mitigate the problem of decision corrosion or design decay in software systems. The key idea of our approach is to translate design decisions or constraints formally specified in OCL to runtime constraints checking code written in AspectJ. Our aspect-based approach has several advantages over other assertion-based approaches. For example, the constraint checking logic is completely separated from the implementation modules, and the implementation modules are oblivious of the constraint checking code, even its existence. Thus, constraints checking code can be easily added or removed from an implementation without modifying the source code. This will enable runtime checks to be applied to different implementations of the same design and also selectively enabled or disabled, for example, for production code.

Our case study on the OCL standard collections library confirmed the feasibility and effectiveness of our approach. We were able to translate most of the OCL constraints to AspectJ constraint checking code. The translation was almost mechanically. By applying the translated AspectJ constraints checking code to a production implementation of the collection library, we were able to detect several errors and deficiencies in both the implementation and the standard specification itself. For example, the standard OCL specification has several collection operations with missing or weak preconditions. We also identified problems in OCL itself, such as unclear semantics for specification inheritance, and issues related with translating OCL constraints to AspectJ code, such as object equality and side-effect freeness of expressions.

ACKNOWLEDGMENT

Cheon and Avila were supported in part by NSF grants CNS-0509299 and CNS-0707874, and all authors were supported in part by the Homeland Protection Institute under Contract No. W9113-08-C-0010 (U.S. Army SMDC).

REFERENCES

[1] AspectJ Project, Available from http://www.eclipse.org/aspjectj/ (last

retrieved on Oct. 2, 2009).

[2] C. Avila, G. Flores, and Y. Cheon, “A Library-based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking,” International Conference on Software Engineering Research and Practice, July 14-17, 2008, Las Vegas, Nevada, pp. 403–408, 2008.

[3] K. Beck and E. Gamma, “Test Infected: Programmers Love Writing Tests,” Java Report, vol. 3, no. 7, pp. 37–50, 1998.

[4] L. C. Briand, W. J. Dzidek, and Y. Labiche, “Instrumenting Contracts with Aspect-oriented Programming to Increase Observability and Support Debugging,” Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, September 25-30, 2005, pp 687–690, Sept. 2005.

[5] A. W. Brown, “Model Driven Architecture: Principles and Practice,” Software and System Modeling, vol. 3, no. 4, pp. 314-327, Dec. 2004.

[6] Y. Cheon, “Automated Random Testing to Detect Specification-Code Inconsistencies,” Proceedings of the 2007 International Conference on Software Engineering Theory and Practice, July 9-12, 2007, Orlando, Florida, U.S.A., pp. 112-119.

[7] Y. Cheon and G. T. Leavens, “A Simple and Practical Approach to Unit Testing: The JML and JUnit Way,” in ECOOP 2002 — Object-Oriented Programming, 16th European Conference, M´aalaga, Spain, Proceedings, ser. Lecture Notes in Computer Science, B. Magnusson, Ed., vol. 2374. Berlin: Springer-Verlag, pp. 231–255, June 2002.

[8] Y. Cheon and G. T. Leavens, “A Contextual Interpretation of Undefinedness for Runtime Assertion Checking,” AADEBUG 2005, Proceedings of the Sixth International Symposium on Automated and Analysis-Driven Debugging, Monterey, California, September 19–21, 2005, pp. 149–157. ACM Press, Sept. 2005.

[9] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, “Model Variables: Cleanly Supporting Abstraction in Design By Contract,” Software Practice & Experience, vol. 35, no. 6, pp. 583-599, May 2005.

[10] D. Coppit and J. M. Haddox-Schatz, “On the Use of Specification-based Assertions as Test Oracles,” Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop, pp. 305–314, Apr. 2005.

[11] B. Demuth and C. Wilke, “Model and Object Verification by Using Dresden OCL,” Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, pages 687-690, July 25-31, Ufa, Russia, 2009.

[12] W. J. Dzidek, L. C. Briand, and Y. Labiche, “Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java,”

ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, October 2-7, 2005, vol. 3844 of LNCS, pp. 10–19. Springer-Verlag, 2006.

[13] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Aug. 1999.

[14] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka, “Overview and evaluation of constraint validation approaches in Java,” ICSE ’07: Proceedings of the 29th International Conference on Software Engineering, pp. 313–322. IEEE Computer Society, 2007.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[16] A. Hamie, “Translating the Object Constraint Language into the Java Modeling Language,” Proceedings of the ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14 -17, 2004, pp. 1531–1535, 2004.

[17] R. Hennicker, H. Hussmann, and M. Bidoit, “On the Precise Meaning of OCL Constraints,” A. Clark and J. Warmer, editors, Object Modeling with the OCL, vol. 2263 of LNCS, pp. 69–84. Springer-Verlag, 2002.

[18] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” Commun. ACM, vol. 12, no. 10, pp. 576–580,583, Oct. 1969.

[19] H. Hussmann, B. Demuth, and F. Finger, “Modular Architecture for a Toolset Supporting OCL,“ A. Evans, S. Kent, and B. Selic, editors, UML 2000 — The Unified Modeling Language, Advancing the Standard, York, UK, October 2000, vol. 1939 of LNCS, pp. 278–293. Springer-Verlag, 2000.

[20] JUnit, Available from http://www.junit.org (last retrieved on Oct. 2, 2009).

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of AspectJ,” J. L. Knudsen, editor, ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest, Hungary, vol. 2072 of LNCS, pp. 327–353. Springer-Verlag, Berlin, June 2001.

[22] K. Kiviluoma, J. Koskinen, and T. Mikkonen, “Run-time Monitoring of Architecturally Significant Behaviors Using Behavioral Profiles and Aspects,” ISSTA ’06: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 181–190, New York, NY, USA, 2006. ACM.

[23] G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary Design of JML: A Behavioral Interface Specification Language for Java,” ACM SIGSOFT Soft. Eng. Notes, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[24] B. H. Liskov and J. M. Wing, “A Behavioral Notion of Subtyping,” ACM

Trans. Prog. Lang. Syst., vol. 16, no. 6, pp.1811–1841, Nov. 1994.

[25] T. O. Meservy and K. D. Fenstermacher, “Transforming Software Development: An MDA Road Map,” IEEE Computer, vol. 38, no. 9, pp. 52-58, Sep. 2005.

[26] B. Meyer, “Applying Design by Contract,” Computer, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[27] OMG, UML 2.0 OCL Specification. Object Management Group, Oct. 2003. Available from http://www.omg.org/docs/ptc/03-10-14.pdf (retrieved on Oct. 15, 2008).

[28] D. E. Perry and A. L. Wolf, “Foundations for the Study of Software Architecture,” ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pp. 40–52, 1992.

[29] D. K. Peters and D. L. Parnas, “Using Test Oracles Generated from Program Documentation,” IEEE Transactions on Software Engineering, vol. 24, no. 3, pp. 161–173, Mar. 1998.

[30] M. Richters and M. Gogolla, “Aspect-oriented Monitoring of UML and OCL Constraints,” The 4th AOSD Modeling with UML Workshop, San Francisco, CA, October 20, 2003, 2008. Co-located with UML 2003.

[31] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, second edition, 2004.

[32] Software Technology Group at Technische Universität Dresden, Dresden OCL Toolkit. Available from http://dresden-ocl.sourceforge.net (last retrieved on Oct. 2, 2009).

[33] J. Warmer and A. Kleppe. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, second edition, 2003.