

An Aspect-Based Approach to Checking Design Constraints at Run-time

Yoonsik Cheon, Carmen Avila, Steve Roach, Cuauhtemoc Munoz,
Neith Estrada, Valeria Fierro, and Jessica Romo

TR #08-38

November 2008; revised February 2009.

Keywords: design constraints, runtime checking, class invariants, pre and postconditions, aspect-oriented programming, Object Constraints 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.

A short version of this report will appear in *ITNG 2009: 6th International Conference on Information Technology, April 27-29, 2009, Las Vegas, NV*.

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

An Aspect-Based Approach to Checking Design Constraints at Run-time

Yoonsik Cheon, Carmen Avila, Steve Roach, Cuauhtemoc Munoz,
Neith Estrada, Valeria Fierro, and Jessica Romo
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968

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: runtime checking, class invariants, pre and post-conditions, Object Constraints Language, AspectJ.

1 Introduction

One of the problems associated with software development and maintenance is *design corrosion* [16] or *design decay* [5]. In general, the corrosion of software design is proportional to the development and maintenance time when an initial design of software gets tweaked to accommodate new or changed requirements or to correct defects. 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 even being noticed by software developers or maintainers. In short, even with rigorous development and maintenance, software tends to lose its original design and becomes difficult to understand and modify.

The Object Constraint Language (OCL) [19] is a textual

notation to specify constraints or rules that apply to UML models such as class diagrams [18]. 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 such as class diagrams. Using the 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 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 many problems such as inconsistency during development and maintenance. For example, as design constraints are not explicitly expressed in source code, a change to source code that causes a deviation from the initial design may not be detected or noticed by the developer. 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 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 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, there should be no side-effects by the

checking code on the states of the program. The other two requirements are of practical concerns. 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. The constraint checking should be plug-and-playable so that it can 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 are separate from the design implementation. AspectJ [11] 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 oblivious to the constraint checking aspects. However, when compiled with the aspects, it will be checked for OCL constraints at appropriate execution points at run-time. Thus, our approach is modular and plug-and-playable.

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 discuss several interesting issues and problems that we encountered during our study. We conclude our paper with related work in Section 5.

2 Background

2.1 UML and OCL

The Unified Modeling Language (UML) [18] 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 relationships. 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

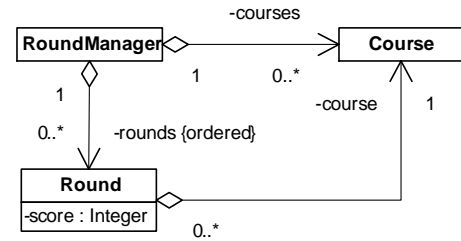


Figure 1. A round manager application

on the diagrams will be incorrect. Thus, there is a need for describing additional constraints on the objects and entities present in the model.

The Object Constraint Language (OCL) [19] 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))
  
```

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. 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.

2.2 AspectJ

AspectJ [11] 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 cross-cutting 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 [11]. 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.

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 responsible 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 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. In the following subsection we show a sample constraint checking aspect.

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
```

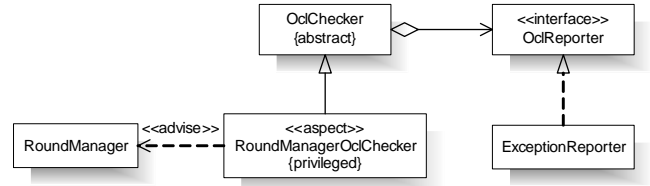


Figure 2. Framework for checking constraints

```
public privileged aspect RoundManagerOclChecker
    extends OclChecker {

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

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

Figure 3. Constraint checking aspect

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 `coursesExecution` 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.

In the next two subsections we explain how we translate OCL constructs and expressions to 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 [19, page 42]. 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 field objects. This kind of state changes cannot be captured using the `set` pointcut because they occur without assignments 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 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 is inefficient.

To remedy the aforementioned problems, we follow the Design by Contract principle [14] 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 constructorExecution(RoundManager m):
    this(m) && execution(RoundManager.new(..));

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

before(RoundManager m): methodExecution(m) {
    checkAllInv(m);
}
```

```
after(RoundManager m): constructorExecution(m)
|| methodExecution(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 `methodExecution`, 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 4 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) 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) {
    checkPre(m.courses.contains(r.getCourse()));
    List<Round> oldRounds = new ArrayList<Round>();
    oldRounds.addAll(m.rounds); // rounds@pre
    proceed(m, r);
    oldRounds.add(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 4 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.

```
after(RoundManager m): constructorExecution(m) {
  checkInit(m.courses != null && m.courses.isEmpty());
}
```

Definition, derivation, and operation body The **def** construct introduce 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 introduce 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 roundsCourseExecution(RoundManager m, Course c):
  this(m) && args(c)
  && execution(List<Round> RoundManager.rounds(Course));

after(RoundManager m, Course c) returning (List<Round> l):
  roundsCourseExecution(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, 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. The OCL expression should be translated to a block of code such as the following, where r contains the result [3]:

```
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 operations 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* (\wedge) operator 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 Discussion and Future Work

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 is silent about the inheritance of constraints [15]. However, for a subclass object to behave like a superclass object

[13], 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+. * (. .))`; remember 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 declaring explicitly 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 [8]. It should be noted, however, that our way of translating pre and postconditions produces the effect of conjoining inherited pre and postconditions, respectively¹; this semantic interpretation is called a *partial exception correctness* [8]. 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 is not clear about the exact semantics of pre and postconditions [15] regarding program termination. There are two choices. First, we can consider the postcondition only 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* [9]; 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

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

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., pointcut `!cflow(within(*OclChecker))` should be conjoined to the constraint pointcuts.

5 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 [10]). Constraints may also be mapped to executable assertions if the implementation language provides a facility such as the `assert` macro or statement [1, 7]. Below we discuss previous work known to us that utilized aspect-oriented techniques.

Briand, Dzidek, and Labiche described an approach for automatically instrumenting OCL constraints in Java using AspectJ [2, 4]. 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 method 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 method. This leads to unnecessary performance overhead. Worse, the approach doesn't work for Java interfaces. 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 [17]. 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 meth-

ods 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* [1]. 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.

Kimmo Kiviluomaa, Johannes Koskinen and Tommi Mikkonen presented an aspect-oriented approach for monitoring the execution of a program using UML behavioral profiles and AspectJ [12]. 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.

Froihofer et al. reviewed and evaluated different constraint validation approaches for Java [6]. They discussed handcrafted approaches, code instrumentation using OCL and JML, 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.

Acknowledgment

Cheon and Avila were supported in part by NSF grant CNS-0509299 and all authors were supported in part by Homeland Protection Institute, Ltd/US Army Space and Missile Command.

References

- [1] C. Avila, G. Flores, and Y. Cheon. A library-based approach to translating OCL constraints to JML assertions for runtime checking. In *International Conference on Software Engineering Research and Practice, July 14-17, 2008, Las Vegas, Nevada*, pages 403–408, 2008.
- [2] L. C. Briand, W. J. Dzidek, and Y. Labiche. Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, September 25-30, 2005*, pages 687–690, Sept. 2005.
- [3] Y. Cheon and G. T. Leavens. A contextual interpretation of undefinedness for runtime assertion checking. In *AADEBUG 2005, Proceedings of the Sixth International Symposium on Automated and Analysis-Driven Debugging, Monterey, California, September 19–21, 2005*, pages 149–157. ACM Press, Sept. 2005.
- [4] W. J. Dzidek, L. C. Briand, and Y. Labiche. Lessons learned from developing a dynamic OCL constraint enforcement tool for Java. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, October 2-7, 2005*, volume 3844 of *LNCIS*, pages 10–19. Springer-Verlag, 2006.
- [5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.
- [6] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka. Overview and evaluation of constraint validation approaches in Java. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 313–322. IEEE Computer Society, 2007.
- [7] A. Hamie. Translating the Object Constraint Language into the Java Modeling Language. In *Proceedings of the ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14 -17, 2004*, pages 1531–1535, 2004.
- [8] R. Hennicker, H. Hussmann, and M. Bidoit. On the precise meaning of OCL constraints. In A. Clark and J. Warmer, editors, *Object Modeling with the OCL*, volume 2263 of *LNCIS*, pages 69–84. Springer-Verlag, 2002.
- [9] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580,583, Oct. 1969.
- [10] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 — The Unified Modeling Language, Advancing the Standard, York, UK, October 2000*, volume 1939 of *LNCIS*, pages 278–293. Springer-Verlag, 2000.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *LNCIS*, pages 327–353. Springer-Verlag, Berlin, June 2001.
- [12] K. Kiviluoma, J. Koskinen, and T. Mikkonen. Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 181–190, New York, NY, USA, 2006. ACM.
- [13] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.
- [14] B. Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, Oct. 1992.
- [15] 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).
- [16] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
- [17] M. Richters and M. Gogolla. Aspect-oriented monitoring of UML and OCL constraints. In *The 4th AOSD Modeling with UML Workshop, San Francisco, CA, October 20, 2003*, 2008. Co-located with UML 2003.
- [18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 2004.
- [19] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, second edition, 2003.