Stirred but not Shaken: Applying Constraints in Object-Oriented Systems

Ragnhild Van Der Straeten, Miro Casanova
Vrije Universiteit Brussel
System and Software Engineering Lab
Pleinlaan 2, 1050 Brussels, Belgium
{rvdstrae|mcasanov}@vub.ac.be

Abstract.

The modelling of object-oriented systems largely amounts to the indication of constraints, representing the knowledge of the domain under consideration. In state-of-the-art case-tools we observe a lack of support for the declaration and implementation of those constraints. Our approach is to express constraints declaratively using the Object Constraint Language (OCL) and to automatically insert constraint checking code in the application. We classify class invariants expressed in OCL according to the navigation paths occurring in these invariants. For each type in this classification a pattern is specified indicating when and how the invariants can be checked. We use Aspect-Oriented Programming for inserting the constraints in the application code. We developed an interactive tool that generates aspect templates using path analysis of the OCL invariant and some coding conventions. Our approach improves the reuse and maintenance of the constraints and the corresponding application.

1 Introduction

Every object-oriented application tries to solve a problem in a given domain. Thus it is normal to think that in order for the application to solve a problem, it must somehow capture and reason with the knowledge of that domain. Using current modelling languages such as the Unified Modeling Language (UML) [UML] is often not enough for expressing the required degree of domain knowledge. We can improve the capture of static aspects of the domain knowledge by using the concept of constraints.

Constraints are defined as restrictions on the entities and the relationships of an object-oriented conceptual model. Therefore, they capture part of the domain knowledge that cannot be expressed with just a model or a set of diagrams. Almost every object-oriented analysis and design method mentions the concept

of constraints. Nevertheless, there is only little support in current case-tools for expressing or checking them in real object-oriented systems.

In current practices we observe that, at the design and specification levels, constraints are expressed in natural language, while at the implementation level, they are manually hard coded with the rest of the application code. This approach results in two major disadvantages:

- The manual translation of the constraints is error-prone, and moreover it makes maintenance of the constraints difficult, because if, for instance, a constraint that is used in several places of the application code has to be modified, then we must also make modifications at each of those places.
- This translation is the only version of the constraints that is neither ambiguous nor informal, as opposed to the natural language version. But, the inherent declarative nature of the constraints is lost. This also makes the constraints hard to review for domain experts.

We can overcome the difficulties mentioned above by having the constraints about entities of a given object-oriented model expressed in a high-level, declarative and precise manner. It is possible to achieve the previous by writing the constraints in the Object Constraint Language (OCL) [KW99] and having the model in UML. Indeed, OCL is a constraint language that comes together with UML, which is meant for expressing constraints in the way we described.

Our intention is to implement the declaratively specified constraints in an object-oriented system, i.e. making the constraints operational. We want to support the developer by providing him/her with a set of patterns. The knowledge of where and how constraints should be checked in the application code is captured in those patterns. Our approach uses aspect-oriented programming (AOP) [GK97] techniques for triggering the constraints. As a consequence, both constraints and application code are separated one from each other.

In section 2 we explain the main issues of checking constraints in object-oriented systems, and we show how and why OCL is used in this context. Also in this section a classification of constraints will be shown. In section 3 we explain the use of AOP and how each of the types of the classification is mapped into a pattern, and how these patterns are implemented by building aspect templates. In section 4 a prototype tool that has been built for generating aspect templates will be briefly explained. This tool uses OCL and generates an aspect template in AspectJ [GK01] that can be inserted into Java code. Finally, we conclude and present some future work.

2 Constraints in Object-Oriented Applications

Constraints in object-oriented design capture domain knowledge to be satisfied by the software application and as such express certain information about the design models. Different ways of expressing constraints can be used. For instance, we can use natural language to express constraints, but it has the drawback that

it is ambiguous, which makes it inadequate for writing the constraints down at implementation time. Also, a constraint language can be used to express the constraints precisely, but the approach has the drawback that the software developers must be acquainted with the formal notation of this language (which is not commonly the case). However, in this paper OCL is used because it is a constraint language that is readable, understandable, usable and yet formal.

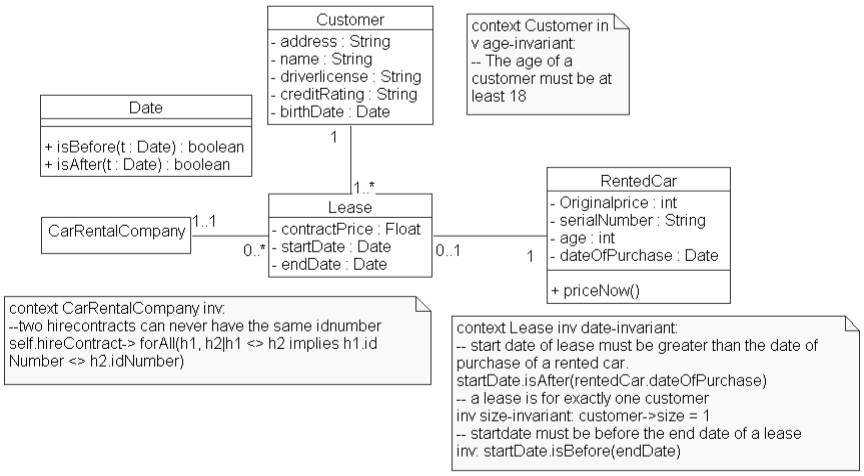


Fig 1. Rent-a-Car Companies.

The example UML class diagram together with some OCL constraints in figure 1 serves as the running example throughout this paper. This model shows a part of a computer system for rent-a-car companies. The constraints are written according to the syntax of OCL1.3 [OCL1.3].

We focus on one specific kind of constraints: class invariants (thus from now on we will use these concepts indistinguishable) and how they can be implemented in object-oriented systems, i.e. how they can be made operational. The reason why we only consider class invariants will be explained in subsection 2.1.

2.1 Making the Constraints Operational

Research on how constraints expressed on a data model can be made operational received a lot of attention in the database world ([MP91],[CW90],[UD91]). From the mentioned research, the research in the object-oriented domain, including our own experience, we observe that the most important issues or difficulties that appear while making constraints operational are:

- *How to translate a constraint into code?*,
- *What to do if a constraint is violated?* and

- *When to check a constraint?.*

The first question is still ongoing work, in the sense of generating code from the constraints. Nevertheless, there are some groups that have been working in generating functional code from OCL constraints. For instance, at the University of Dresden [HDF00] an OCL compiler for Java has been developed, which generates some extra methods in the context class of a constraint. Also, a code generator for OCL expressions has been developed in our laboratory but for a specific domain (the geographic data domain) [CDW01].

The second question is related to what we call *rescue*, based on the notion of rescue in Eiffel [BMEi]. If a constraint is violated, several actions can be undertaken. For example, to raise an exception, to print a message to the user of the system, to dump a program trace or even to stop the application. The problem with those actions is that they leave the application in an inconsistent state. However, eventually the programmer would like to specify some actions to enforce the constraints in the application.

In general, there are three possibilities for enforcing constraints. Constraint checking happens before a change to a value of one of the constraint's components is made. If the new value violates the constraint, the change does not happen and the old state is preserved. In this case the violation of the constraint is *avoided* or *forbidden*. Another possibility is to check the constraint after a change has happened. If the constraint is violated due to the change, a *rollback* is performed. This also implies that the old state is preserved. A third possibility is to check the constraint before or after a change and, if it is violated, the state is fixed in such a way that the constraint is preserved anyway. This is also called the performance of some *compensating actions*. In this case the object(s) is(are) in a new state and the constraint is ensured. We are convinced that the rescue of a constraint should be determined by user interaction because there are cases where enforcing a constraint can have no meaning at all, or in other words, it is desirable to "see" that the constraint is being violated. As default an exception can be raised, sending a message to the screen or another interface component.

The last issue has to do with the determination of the *scope* of a constraint, i.e. the points in the application code where the constraint should be checked (called the *insertion points* in this paper). The scope of a constraint is highly related to the types it reasons about, or in other words, it depends on the navigation path of its corresponding OCL expression. As the navigation path becomes longer, more classes become involved and more insertion points have to be specified. These insertion points are scattered over the several classes that can be distilled out of the navigation paths of the class invariant. The complexity of this issue led us to creating a classification of constraints based on their navigation paths, or the classes they involve. We do not address the pre- and post-condition in this paper because the scope of these kinds of constraints is straightforward.

The definition of the scope of the invariant is different from the one in relational databases. In a relational database management system (RDBMS) only insert, delete and update operations can occur affecting the state of the database

[CW90]. In object-oriented systems, the number of possible affecting operations is much larger. The definition of the scope is a lot more fine-grained. Different methods for example can be specified assigning a value to a field of an object. This is also the case in for example the O_2 system in [MP91]. Here the possible sources of constraint violation are derived completely manually.

In the last decennium several papers were written about adding the Design by Contract principle [BM91] to mainstream programming languages. Most of these tools are extensions to the Java programming language ([RK98], [KHB98], [DH98]). A lot of emphasis is put on the implementation of the mechanism used to check the constraints. This is however only one aspect of the entire issue of constraint checking.

2.2 Classification of Invariants

An OCL class invariant is specified as an expression that involves different OCL types. OCL allows next to the specification of *intraclass* constraints, i.e. constraints on the features of one class, also the specification of *interclass* constraints. This last kind of invariants constrains associated classes and their features. The term feature is used to denote attributes and operations of classes and association ends. OCL has a navigation operator, the dot operator, which allows navigation across the different associations connecting the system's classes. This operator allows us to build quite large navigation expressions. Usually, OCL invariants consist of several navigation expressions combined with relational, logical or arithmetic operators. Invariants can be divided into four groups based on the complexity of the navigation expressions used.

The different types of the classification are shown in figure 2. The features the constraint poses some restrictions on are accentuated in the diagrams. In constraint types 2 to 4, the classes on a grey background are intermediate classes. The dotted line under these classes indicates that there may be n intermediate classes with n an integer greater than or equal to zero.

This classification is useful because it is possible to know with it the insertion points of a constraint. For every kind of constraint in the classification, there will be different insertion points depending of course on the navigation paths of the OCL expression. Due to the previous, the classification pretends to give a solution to the problem of the scope. We will show how this concept of classification is actually used in section 3 for the mentioned purpose.

3 Patterns for OCL Invariants

For each type of invariant a pattern has been developed specifying some examples and the different insertion points. Insertion points specify where and how an invariant should be checked. OCL is a pure modelling language and does not take any runtime issues into account. In [KW99] pg. 85 we find *Invariants are defined as being true at any moment in time.* The right balance between code-checking as much as possible and performance should be found. This is however application dependent. In the Design by Contract principle a class invariant is

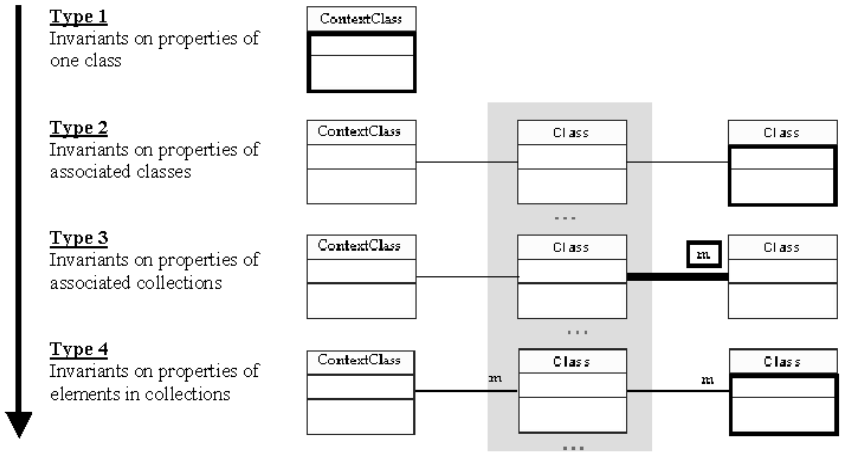


Fig 2. Classification of OCL Invariants.

checked on entry to and return from external calls to the object. This means that before and after the call to a public method and after the execution of the constructors the class invariants must be checked. The insertion points specified in the patterns follow this approach by only considering public methods and coding conventions.

The patterns can be implemented using different techniques. We show how AOP can be used to implement the checking of the different types of invariants. The next subsections specify patterns representing the types of invariants introduced in figure 2. From the pattern aspect templates are generated. The language used is AspectJ [GK01], an aspectual extension of Java developed at Xerox Parc. For each type of invariant an example is shown of the different aspects generated. The aspects are generated using the OCL invariant, the UML class model and coding conventions. How the OCL invariant and the class model are used to fill the pattern is explained in the last subsection.

Getting and setting methods are used for accessing and modifying each feature of a class. For collections of objects, methods to add or delete objects of the collection or replace the whole collection are defined. This boils down to the following methods, where `XXX` stands for the feature name:

1. `getXXX()`: allows to access a property of a class,
2. `setXXX(<type> newXXX)`: allows the modification of a property of a class or to replace a whole collection,
3. `addXXX(<type> newXXX)`: allows the addition of an element to a collection,
4. `deleteXXX(<type> oldXXX)`: allows the deletion of an element from a collection.

3.1 Invariants on Properties of One Class

Examples A reasonable invariant in the rent-a-car company example is to demand that every customer has an age greater than or equal to 18. In OCL, this can be written as:

```
context Customer inv age-invariant:  age >= 18
```

where the context class is `Customer`. Another example constraint in figure 1 could be to demand that the start date of a lease must be before the end date of the lease, which can be indicated in OCL as follows:

```
context Lease inv:
 startDate.isBefore(endDate)
```

Remark that OCL allows the use of user-defined operations on user-defined types.

Insertion Points The insertion points for this kind of invariant are: after the constructors of the context class, before and after all public methods of the context class that access the involved instance variables, before and after all public methods of the context class that change the involved properties. If the involved properties of the class are available to all classes to which the context class is visible, there must be a check before and after all public methods of these classes that access or change these properties. A basic principle of object-oriented programming however is that a class is the only responsible for its data. Therefore the properties of a class should be private. However, in C++ and Java it is possible for an object to access the private properties of other objects of the same class.

Template Aspects For these kinds of invariants, one template aspect is generated for the context class. As an example, consider the template aspect for the `age-invariant`.

```
aspect AspectCustomer of eachobject(instanceof(Customer))
 {
 pointcut constructors(Customer customer): instanceof(Customer) && executions(new(..));
 pointcut setage(int newage): instanceof(Customer)
  && receptions(public void setage(int newage));

 after(Customer customer): constructors(customer){
  if !(assert()){ /* action to perform */ }
 }

 after(int newage): setage(int newage){
  if !(assert()){ /* action to perform */ }
  }

 boolean assert(){ /* implementation of the OCL invariant */ }
 }
```

The insertion points in this case are after the constructors of the `Customer` class and after the `setage` method. This is indicated by the join points `constructors` and `setage` and the advice, i.e. the `after()` statements indicating that a certain piece of code (i.e. the checking of the invariant) should be executed after the code in the join points.

3.2 Invariants on Properties of Associated Classes

Examples This kind of invariant is an extension of the first type. The properties participating in the invariant do not have to belong to the same class but a combination of properties of different classes is possible. As an example consider the invariant that the start date of a lease must be greater than the date of purchase of the associated rented car. In OCL this can be written down as follows:

```
context Lease inv date-invariant:
startDate.isAfter(rentedCar.dateOfPurchase)
```

Remark that the OCL expression can contain more than one navigation. The increasing length of the navigation path increases the number of classes and methods involved in the checking of the invariant.

Insertion Points The following insertion points should be considered: after the constructors of the context class, before and after every public method of the context class and the classes involved in the associations that access or change the involved associations or the involved attributes. Before and after every public method of the classes to which the involved associations and attributes are visible are also insertion points to be considered.

Template Aspects For each class involved a template aspect is generated. For the `date-invariant` the following aspects are generated:

```
aspect AspectLease of eachobject(instanceof(Lease)){
   Lease connectedLease;

   pointcut constructors(Lease lease): instanceof(lease) && executions(new(..));
   pointcut setstartDate(): instanceof(Lease) && receptions(* setstartDate(..));
   pointcut setrentedCar(RentedCar rentedCar): instanceof(Lease) &&
     receptions(* setrentedCar(rentedCar));

   after(Lease lease) : constructors(lease){
      connectedLease = lease;
    }
   after() : setstartDate(){
        if (!assert()){ /* action to perform */ }
  }
   after(RentedCar rentedCar) : setrentedCar(rentedCar){
         AspectRentedCar.aspectOf(rentedCar).setAspect(this);
         if (!assert()){ /* action to perform */ }
  }
  boolean assert(){ /* implementation of the OCL invariant */ }
}

aspect AspectRentedCar of eachobject(instanceof(RentedCar)){
     AspectLease anAspect;

    pointcut setdateOfPurchase(): instanceof(RentedCar) && receptions(* setdateOfPurchase(..));
    pointcut setlease(): instanceof(RentedCar) && receptions(* setlease(..));

    void setAspect(AspectLease anAspectLease){
        anAspect = anAspectLease;
    }

  after(): setdateOfPurchase(){  }
  after(): setlease(){  }
}
```

The insertion points are also here indicated by the different join points corresponding to the modification of the different attributes `startDate, dateOfPurchase` and association ends `lease, rentedCar`.

3.3 Invariants on Properties of Collections

The navigation over several associations results in a collection. Navigation over just one association with multiplicity greater than one results in a set, while navigation over more such associations results in a bag [KW99]. A sequence is obtained when navigating over an association that is adorned with {`ordered`}. On collections certain operations are predefined in OCL.

Examples Xor-, subset- and multiplicity-invariants, specifying restrictions on the size of collections are examples of this kind of constraint. An xor-invariant [KW99] indicates that any instance of the class may only participate in one of the associations at any time. The xor-invariant puts some specific limitations on the participating objects in the associations. A subset-invariant indicates that the set of links for one association is a subset of the set of links for the other association. The addition of objects to the subset and the deletion of objects to the superset should be managed. An example of a xor- and subset-invariant is shown in figure 3. These invariants can be written down as follows:

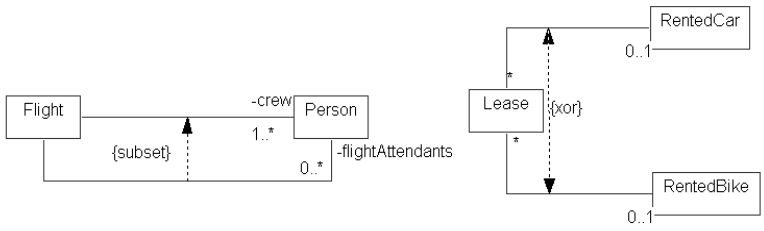


Fig 3. Example of Xor- and Subset-invariant.

```
context Lease xor-invariant inv:
rentedCar->notEmpty xor rentedBike->notEmpty

context Flight subset-invariant inv:
flightAttendants.include(crew)
```

Insertion Points The insertion points to be considered are: after the constructors of the context class, before and after every public method of the context class and the classes involved in the associations that access or change the involved associations. Also before and after every public method of the classes to which the involved associations are visible, must be considered.

It is possible to limit the number of insertion points by looking at the functionality of the OCL operations on the collections. For a subset-invariant only

the methods that add objects to the subset or remove objects from the superset should be investigated. Another OCL operation on collections is the *exists* operation. It can be used if we want to specify that there is at least one object in a collection for which a certain condition holds. We need only to take care of the constraint if elements are removed (or replaced) from the collection.

Template Aspects As an example, consider the `subset-invariant`. Only the join points of the template aspect generated for the class `Flight` are shown below.

```
aspect AspectFlight of eachobject(instanceof(Flight)){

   pointcut constructors(Flight flight): instanceof(flight) && executions(new(..));
   pointcut setflightAttendants(): instanceof(Flight) && receptions(* setflightAttendants(..));
   pointcut setcrew(): instanceof(Flight) && receptions(* setcrew(..));
   pointcut addcrew(Person person): instanceof(Flight) && receptions(* addcrew(person));
   pointcut deleteflightAttendants(Person person): instanceof(Flight) &&
    receptions(* deleteflightAttendants(person);
```

3.4 Invariants on Properties of Elements of Collections

Examples This category contains invariants that deal with the elements inside collections. This means that in order to evaluate these constraints, we will have to iterate over the elements. Consider the invariant expressing that two leases can never have the same identification number within a rent-a-car company.

```
context CarRentalCompany inv identificationinvariant:
self.lease-> forAll(h1, h2|h1 <> h2 implies
             h1.idNumber <> h2.idNumber)
```

Insertion Points The insertion points to be considered are: after the constructors of the context class, before and after every public method of the context class and the classes that access or change the involved associations or the involved properties, before and after every public method of the classes to which the involved associations or properties are visible.

Template Aspects The difference between this type of constraint and the previous one is the participation of properties of the elements of collections. The generated template aspects are analogue to the previous ones, the only difference is that in this case in some aspects the setting methods for the involved properties should appear.

3.5 Filling in the Patterns using the UML Model and the Invariant

The insertion points given in the different patterns are very general. How do we determine the involved classes and methods? The OCL constraint is used to determine the context class and the involved associations, attributes and operations. The associated classes can be determined by using the navigation paths specified by the constraint and by looking up these paths in the UML model, also the attributes and operations specified in the navigation paths can be looked up in the UML and OCL model. In the case of the `date-invariant`, the context class is `Lease`. Using the coding conventions, we observe that the

methods `setstartDate(Date newstartDate)` and `setdateOfPurchase(Date newdateOfPurchase)` changing the involved attributes and the methods `setlease (Lease newlease)` and `setrentedcar(RentedCar newrentedCar)` changing the involved association ends should be monitored.

4 Automated Constraint Checking

An prototype tool has been developed for the translation of declarative invariants to aspects. The invariants are expressed in OCL and written down on detailed design diagrams. The specification of the invariants separately from the code has the advantage that they can be analyzed and optimized without even thinking about a programming language. The embedment of the tool in the Rational Rose environment [Rose] allows the programmer to specify the invariants on a design diagram and to make them operational with a single push on a button. The tool generates template aspects containing the different insertion points and the necessary signatures of aspect methods. A template aspect is produced for each class which instances can affect invariants. The generation of the aspect templates can be divided into four separate parts:

1. The OCL invariants specified on the UML class diagram are written into a file, readable to the parser.
2. The invariants are checked for syntax errors and for compliance with the UML model.
3. The navigation paths are analyzed to find all affecting objects and methods.
4. For each involved method joint points are generated.

The generator expects the OCL class invariants to be written down in notes on the UML class diagram. The text within the notes is expected to contain valid constraints expressed in the OCL version 1.3.

5 Conclusions and Future Work

We observed that the use of constraints nowadays is barely supported by existing case-tools, and constraints are normally hard-coded and mixed together with the rest of the application code. Expressing the constraints in a declarative, high-level and precise manner by using a constraint language (in this case OCL) has been shown to be an important step towards a better solution. Also, by using AOP techniques we add modularity to the constraint, i.e. constraints are separated from the rest of the application code. The advantages of our approach are:

- Improvement of the maintenance and understandability of the constraints due to the use of a high-level and declarative constraint language. Every

time a constraint has to be changed, only its OCL specification needs to be modified.

- Improvement of the maintenance of the application code because the constraint code is not mixed with it.
- Increased independence of the evolution of the constraints and the application code. They are linked just by the insertion points.

We also showed a classification of constraints that can be helpful to the designer and programmer while creating an object-oriented system. Every type of that classification is mapped to a pattern which indicates the insertion points. Finally, we implemented a prototype tool that generates aspect templates automatically in AspectJ from the OCL specifications of the constraints.

Some areas of future work have been identified. The path analysis can still be refined taking into account the operations defined by OCL. The set of allowable constraints can be extended. Support can be given to the pre- and postconditions. A mechanism enabling and disabling the constraints can be installed. The integration in an UML case-tool can be extended by allowing the developer to specify the events, the actions and the enable/disable function at the model level instead of changing the code of the aspects.

References

[KW99] A. Kleppe, J. Warmer. "The Object Constraint Language: Precise Modeling with UML". Addison-Wesley, 1999.

[HDF00] H. Hussmann, B. Demuth, F. Finger. "Modular Architecture for a Toolset Supporting OCL". Proceedings of UML 2000. Springer-Verlag. York, UK, October 2000.

[CDW01] M. Casanova, M. D'Hondt, T. Wallet. "Explicit Domain Knowledge in Geographic Information Systems". Proceedings of SEKE 2001. Bs. Aires, Argentina, June 2001.

[BM91] B. Meyer. "Object-Oriented Software Construction". Prentice Hall. 1991.

[BMEi] B. Meyer. "Eiffel, The Language". Prentice Hall. 1992.

[GK97] Gregor Kiczales et all. "Aspect-Oriented Programming". Proceedings of ECOOP '97. Springer-Verlag. Jyvskyl, Finland. 1997.

[GK01] Gregor Kiczales et all. "An Overview of AspectJ". To appear in proceedings of ECOOP '01. Springer-Verlag. Budapest. Hungary. 2001.

[MP91] C. Medeiros, P. Pfeffer. "Object Integrity Using Rules". Proceedings of ECOOP '91. Springer-Verlag. Geneva, Switzerland. 1991.

[KHB98] M. Karaorman, U. Holzle, J. Bruno. "jContractor: A Reflective Java Library to Support Design By Contract". Technical Report TRCS98-31. 1998

[DH98] A. Duncan, U. Holzle. "Adding Contracts to Java with Handshake". Technical Report TRCS98-32. 1998.

[RK98] R. Kramer. "iContract: The Java Design By Contract Tool". Proceedings of Tools USA '98. IEEE Computer Society. Santa Barbara, California. 1998.

[UD91] S. Urban, M. Desiderio. "Translating Constraints to Rules in CONTEXT: a Constraint Explanation Tool". Proceedings of Object-Oriented Databases: Analysis, Design and Construction '91. Elsevier Science Publishers B.V. 1991.

[CW00] S. Ceri, J. Widom, R. Cochrane. "Practical Applications of Triggers and Constraints: Successes and Lingering Issues". Proceedings of the 26th VLDB. 2000.

[CW90] S. Ceri, J. Widom. "Deriving Production Rules for Constraint Maintenance". Proceedings of the 16th VLDB. pages 566-577. 1990.

[Rose] The Rational Website. http://www.rational.com/.

[UML] I. Jacobson, G. Booch, J. Rumbaugh. "The Unified Modeling Language Reference Manual". Addison-Wesley. 1999.

[OCL1.3] OMG. "Object Constraint Language Specification, version 1.3". OMG document ad990609. http://www.omg.org/ . 1999.