



Research Center

NRC-TR-2006-006

Using Class Relationships for Identifying Invariants

Ian Oliver
Nokia Research Centre Helsinki
<http://research.nokia.com>
17 May 2005
ian.oliver@nokia.com

Abstract

Formally specified invariants on UML models greatly enhance the correctness of the model by restricting the state space such that invalid scenarios that are admitted by the class diagram can be forbidden. Identification of these invariants in the situations where objects may or may not be shared is critical to the development of the model. Simple patterns based upon “shapes” or patterns that exist in the class diagram can be applied to assist the developer in finding suitable places for these invariants and thus improving the preciseness or rigor of the models and the correctness of the system under development.

Index Terms

UML
Class Diagram
Formal Specification
Design Pattern
Invariants
OCL

1. Introduction

In response to the complexity of constructing models and the identification of commonly recurring scenarios or idioms we have seen an explosive growth in the number of patterns. One area that has not yet benefited from this is the formal development of models. Some work is proceeding in this area but is currently focusing on languages such as Z [13].

The UML [9] can be used in formal development [2] and does have features that make it suitable for the expressing of constructs such as pre/post-conditions and invariants. The UML contains an integrated, textual constraint language: The Object Constraint Language (OCL) [14] for precisely this purposes. There however exist few patterns that address the usage of OCL in general usage; the manipulation and specification of business rules specified in OCL [5] and the temporal specification of dynamic systems [15]. Notably in the Catalysis method this pattern for identifying constraints from circularities was presented in a limited form .

In the PUSSEE project [6] a tool was developed for translating UML models to the B specification languages for verification through theorem proving. While it is very easy to construct class diagrams that are very underconstrained (and easily provable) this results in systems that admit (accidentally) incorrect situations. Even when various rules suggested by the theorem prover or by the business case were added it was still the situation that the proven system could still admit incorrect scenarios.

To address this we identified a number of techniques where one could simply analyse the shape or structure of the class diagram to identify likely places for the existence of invariants. The bulk of these invariants addressed specifically the sharing of objects which were often taken to be implicitly specified (and not always in the same way by different engineers).

In this paper we present a discussion of how one models with invariants in the UML through the use of a simple case study. We then introduce simple patterns for the identification of places where invariants are required based upon object sharing. Finally we discuss some additional UML constructs and their relationships with invariants.

2. Modelling with Invariants

When modelling with graphical languages such as UML - to which this paper is addressed - it is necessary to supplement the graphical class diagrams with textual constraints to capture information that can not be expressed graphically. In the case of the UML this is done by utilising the Object Constraint Language (OCL).

OCL is tightly integrated into the UML and contains features for navigating across models through association roles to target sets of objects. It also contains large set of operators (union, intersection, selection etc) for various collection types (sets, sequences etc). The logic system is based on a three valued logic where the undefined value is used for cases where an expression can not be evaluated: empty sets, attribute values not set, private elements. The undefined value is not explicitly exposed unlike true and false but tested using an inbuilt OCL operator.

Statements in OCL are written in the context of a class and reference to particular instances of that class (ie: objects) are made through the special variable `self` which behaves analogously to similar keywords found in most OO languages (this, self, me etc). OCL can be used to specify guards on state transitions, actions in terms of pre/post conditions and class invariants. It is this latter structure that we are interested in and this takes the form:

```
context "Class Name"  
inv: "OCL Expression"
```

Invariants must always resolve to True given a correctly formed object diagram [7]; that is one that satisfies both the class diagram specification and the syntactical and semantic rules of UML. If an invariant resolves to false then the scenario is then deemed to be inconsistent or invalid.

In figure 1 we show a simple class diagram. Here a library has a catalog of books, each of which has a number of copies (we allow the fact that there might not be any copies of a book!), a number of members and a way of keeping track of current and historical loans of those members.

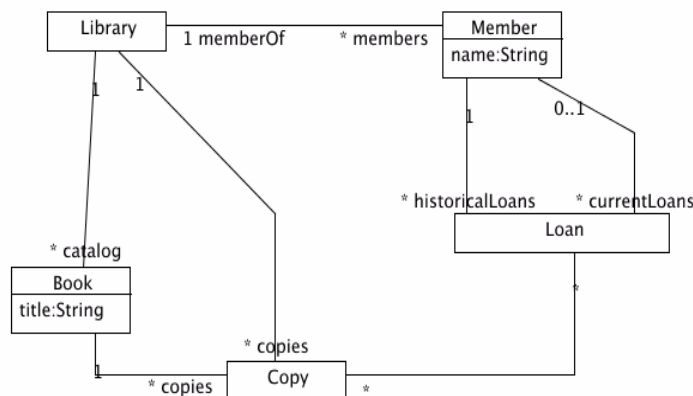
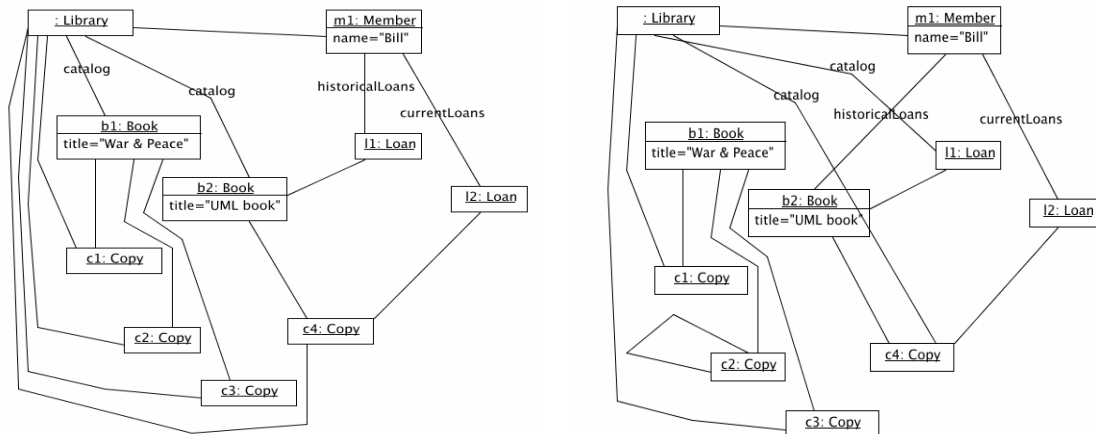


Figure 1: Example Class Diagram of a Liibrary System

In order to validate this model we can create any number of object diagrams showing particular scenarios. These scenarios may be generated by hand or through some techniques such as animation/simulation [8][10]. These object diagrams must conform to the given class diagram; unless of course we are deliberately showing erroneous or impossible scenarios. These scenarios themselves can be hand coded, automatically generated or produced by some means of execution.

In figure 2 are shown two such examples, the first shows a scenario where everything is fine and the second where there are deliberate mistakes which can not occur because these are forbidden by the class diagram.



Valid or Correct Object Diagram

Invalid Object Diagram

Figure 2: Two Object Diagrams Showing Different Scenarios

The “invalid object diagram” is syntactically correct UML but there are a number of problems: there is a link to self with object c2, object b1 does not have a link to the library object, a link with role catalog exists between the library object and object l1 - all of these are illegal with respect to the given class structure shown in figure 1.

Both of these diagrams have been generated with some domain assumptions. One thing that is often assumed is that the sharing of objects is implicitly specified such that whether objects can be shared or not across associations is somehow “known” without it being explicitly stated, or in other words that the class diagram is somehow “magically” neither under or over constrained. This is a major source of errors which are either not captured or difficult to capture during testing - many tests assume the model is correct in the first place!

We have not specified so much information in the given model - at least so far only the class structure. Let us now introduce the following “business rule”:

Members can not borrow books from other libraries

which can be specified as an invariant in OCL as:

context Member

```

inv: self.currentLoans→forall(1 | 1.copy.library =
self.memberOf)

```

To demonstrate this the scenario shown in the object diagram in figure 3 obviously breaks this. Here the member denoted by object m1 has a current loan to copy “c4” which belongs to

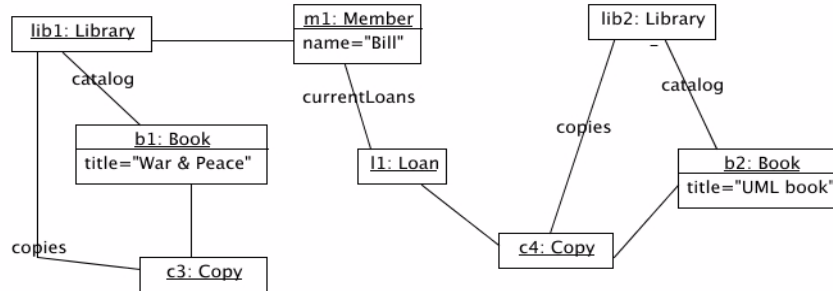


Figure 3: Member Loaning from a Different Library

to (or is one of the copied held by) library “lib2” - this is obviously not the same library that the member “m1” is a member of, hence this particular model does not conform to the given invariant.

While application of business rules such as these constrain the model they still do not tackle the problem of missed invariants due to sharing and indeed it is often the case that these sharing invariants are *not* fully captured by the set of business rules. For example we have a case such that copies can belong to different libraries as shown in figure 4. Indeed this case does require some explaining. In the class diagram which defines the set of valid object diagrams (fig.1) it is stated that a book “belongs to” one library as indeed object b1 does; a book has one or more copies (ie: object c1) and that a library can have many copies as object lib2 does via the role copies to object c3; thus this object diagram is correct with respect to the class diagram. Note, that for clarity we have left out links which are superflu-

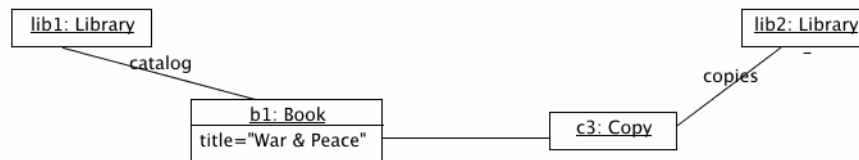


Figure 4: A Copy Belonging to the Wrong Library

ous to this example.

A similar case exists between Members and Loans where it is obvious that the sets of loan objects reachable by navigating the historicalLoans and current Loans should be disjoint. However in many cases this is often left unspecified and situation in figure 5 can occur.

Again an explanation: member m1 has made 3 or 4 (depending on whether we are counting loan objects or relationships between the member and loan) loans. Over time the loans have been for the same particular copy of a book - this is permitted of course. However the current loan also co-incides with one of the historical loans; this is a case which should not occur. This particular case should be apparent and protected against by a business rule, it is also apparent that the potential sharing of loan object is possible. The invariant below prevents this situation:

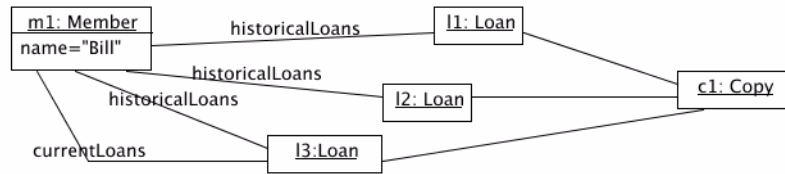


Figure 5: Non-disjoint Historical and Current Loans

context Member:

inv: $(\text{historicalLoans} \rightarrow \text{intersection}(\text{currentLoans})) \rightarrow \text{isEmpty}()$

While it can be argued that a business rule and enough precondition protection on actions on the model will prevent these situations we can demonstrate in various cases that the specification of the "implicit, embedded in the domain" knowledge is not recorded and subtle errors do occur. What is happening is that the *semantics* english language (or natural language) names is somehow implying the disjointedness of these two roles, but as we can see this is not captured in the specification unless the invariant is present.

3. Identifying Invariant Locations through Circularities

In the previous section we have described the problems of missing invariants and the possibilities that not all business rules on the model can capture all the instances of potential object sharing. We present here two basic cases for the identification of locations of invariants based upon object sharing and how to write the associated invariants. We then continue with a discussion of how these basic patterns can be applied to attributes, association classes and composition/aggregation.

3.1 Sharing

Sharing is simply identified by there existing two or more relationships between any pair of classes. The situation seen in figure 5 is a classic example of this. This scenario is easily identified by situations where two or more relationships are between the same pair of classes as shown in figure 6. All cases can be reduced to a pair of relationships implying an invariant of the form:

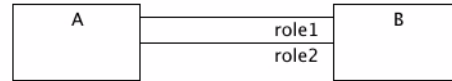


Figure 6: Sharing via 2 role

context A:

inv:`self.role1ToB→select(...)` φ `self.role2ToB→select(...)`

where φ defines the relationship between two sets of objects. In the example earlier this was simply the assertion that the intersection of the two sets of objects is always empty, ie: disjoint sets.

The situation where three or more relationships exist as shown in figure 7 is reduced to all the combinations of the relationships. In the case of just three there are three possible invariants resulting in an invariant of the form:

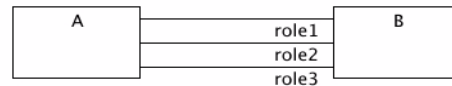


Figure 7: Sharing via 3 roles

context A:

inv:`self.role1ToB→select(...)` φ `self.role2ToB→select(...)`

and `self.role1ToB→select(...)` φ `self.role3ToB→select(...)`

and `self.role2ToB→select(...)` φ `self.role3ToB→select(...)`

This invariant is then in the extreme form where everything is made explicit, further analysis might reveal that the invariant can be simplified somewhat depending upon the nature of the relationships.

3.2 Class Circularity

This is in effect a generalisation of the sharing pattern and manifests itself by identifying circularities between three or more classes. A circularity always implies that information is required about whether the objects in the circle remain closed or open; the case shown in

the previous erroneous object diagram in figure 4 was an example where the Library-Book-Copy circularity was left open.

In the library model the only 3-class circularity exists as shown in figure 8 between the said classes; as we have said, a circularity implies an invariant which in this case closes the sharing.)

```
context Library
inv: self.catalog.copies=self.copies
```

In the more general case where we have classes A, B and C related in a triangle (analogous to figure 8) with suitable roles on the associations between them:

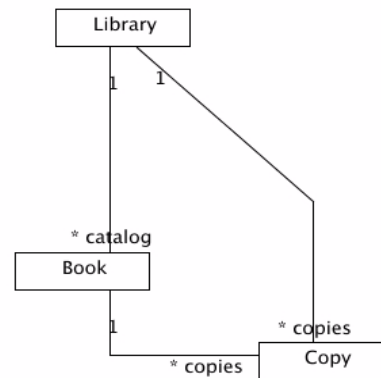


Figure 8: Library, Bank and Copy Circularity

```
context A
inv: self.roleToB.roleToC→select(...) φ
    self.roleToC→select(...)
```

which states that choosing one class A, the set of objects of class C which are derived by a navigation from A via B to C has some relationship (defined by an operator ϕ) to the set of objects of class C which are derived by a navigation from A to C. There may be some filters as defined in the select statement on the sets of objects if necessary.

Larger circularities can also be treated in this way although with invariants of the form for 4-role circularities:

```
context A
inv: self.roleToB.roleToC.roleToD→select(...) φ
    self.roleToD→select(...)
```

In the case of the 3-role circularity any potential starting point (A,B or C; Library, Book or Copy) results in the same invariant albeit written in slightly different forms because of the context. In the case of larger circularities some starting contexts result in differing kinds of invariant.

5-role invariants and larger are possible but the information contained in the invariant generally tends to be of lesser value and is anyway normally captured by the smaller circularities. In all cases with 4 or more roles, these cases can be reduced to combinations of 3-role invariants where the set of objects described by 2 or more roles is coalesced into one set. This is analogous to the splitting of any 4 sided or more polygon into two or more internal triangles.

In figure 1 there are two 5-role circularities which we can investigate using this technique; for simplicity we choose only the route from Library via the roles members, currentLoans,

copy, book and library. Writing the invariants across this in the form given above for 4 or more roles and then changing the contexts is normally difficult and it is easier to consider this in terms of its constituent “triangles”. For example, if we take the triangle Library-Member-Loan (via currentLoans) we need to derive the direct relationships between Library and Loan. This might be achieved by the expressions in the context of Library

- `catalog.copies.loan`
- `copies.loan`

Treating these as atomic constructs then we need to complete the invariants which are in the forms:

```
context Library  
inv: self.members.historicalLoans→select(...)  $\varnothing$   
      catalog.copies.loan →select(...)
```

```
context Library  
inv: self.members.historicalLoans→select(...)  $\varnothing$  copies.loan  
      →select(...)
```

To improve the simplicity of this we assume that we have already found the triangle between Library, Book and Copy earlier which makes both of the above identical.

Now that we have identified the source of object sharing we can decide whether we wish to allow or deny (or restrict in some other selective way) the sharing. In this case it is probably obvious that the loans should be unique to that library and that the set of historical loans should be the same whichever path is taken to them:

```
context Library  
inv: self.catalog.copies.loan  
      →includesAll(self.members.historicalLoans)
```

Note that the one navigation is included in the other meaning that the sets are not necessarily equal in this case. This paper is not concerned with the reasoning of this but if we had asserted their equality then the conjunctions of the sharing invariant between historicalLoans and currentLoans and the invariants found from the 5-role circularity with currentLoans instead of historicalLoans would result in at least an overly constrained model and at worse would not have admitted any object diagrams. Discovery of this may be made by inspection by hand or by theorem proving.

4. Other Constructs

The UML contains other constructs in class diagrams other than the basic class and association. In this section we describe how these constructs either provide additional constraints or can be mapped into the forms seen earlier in this paper and thus be admitted to the technique described.

4.1 Composition and Aggregation

Composition and aggregation are commonly thought of as describing “part-of” type relationships, the reality is more complicated and is more grounded in philosophy. What can be said is that they provide some constraints on sharing of object and in general one can state (although this is certainly open to debate) that the internal parts of some structure can not be shared.

The actual definitions of composition and aggregation are based upon often particular implementation platforms and concepts such as pass-by-value versus pass-by-reference; the UML2.0 specification states the following:

- the multiplicity of the aggregate end (syntactically the diamond) is 0..1
- precise semantics of shared aggregation varies by application area and holder
- composition indicates that the composite object has responsibility for the existence and storage of composed objects (also known as parts)

In this section we restrict ourselves to just dealing with the concept of sharing and the afore stated criteria that composed or aggregated objects can not be shared. In this respect composition and aggregation appear the same and syntactically (and semantically) we only use one concept.

In figure is a simple class diagram describing the composition relationship between Books, Papers and Sections, one can read this diagram as a book is composed of many sections and similarly papers are composed of many sections. However from a sharing point of view sections can not be shared between Books and Papers.



Figure 9: Books, Papers and Sections

In this case it is not necessary to explicitly write an invariant denoting the forbidding of the sharing as this is implied directly in the UML language by the use of the aggregation and composition semantics. However as the UML2 semantics does point out, there is a (sic.) semantic variation point which may potentially allow sharing in certain application areas. Resolution of these semantic variation points is outside the scope of the model being constructed and is the responsibility of the language itself.

4.2 Attributes

Attributes are normally used for situations where the type of the attribute is a value type or some kind of general purpose class type such as Date or Time. Value types present an interesting situation in that the instances of these types are not uniquely identifiable, for example instances of integers are not references by object-id but by the value they contain - one does not think of an expression such as 2+2 as being between two separate instances of an integer class whose “values” happen to be 2; unless one is a SmallTalk programmer.

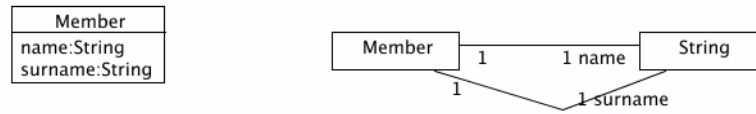


Figure 10: Attributes and Associations

Generally we have found that there is little need to restrict sharing across attributes where value-types are concerned. Consider the case in figure 10 which contains two class diagrams (one with attributes and the other as the attributes expanded out to associations identified by roles). The sharing pattern described in section 3.1 implies the possibility of an invariant between the roles and it is possible to end up with semantic nonsense¹ (in terms of reality - this is perfectly acceptable UML and a perfectly acceptable model) such as

```

context Member:
inv: not(name = surname)
  
```

However, in the cases where the types of the invariants are classes other than utility or singleton classes (these **must be** shared/can not be restricted due to only one instance existing) then the sharing pattern holds.

4.3 Association Classes

When links are created between objects it is sometimes necessary to include additional information characterising the link itself. This is specified on the class diagram by using association classes. When a link of that association is then created an instance of the association class is also created. Semantically the link and the instance of the association class are part of the same entity. One could rewrite the associations between Member and Copy using Loan as an association class.

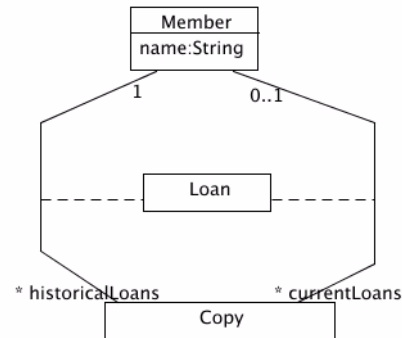


Figure 11: Loan as an Association Class

This model shown in figure 11 is semantically equivalent to the class diagram in figure 1 plus the sharing invariants. However mechanically mapping between the two is not always easy or desirable.

In terms of object sharing an association class' semantics provides the protection necessary and instances of association classes can not be considered in the same way as objects which are instances of classes.

1. This would preclude people such as James James (son of Evan James) of Pontypridd, Wales, writers of the Welsh National Anthem in 1856.

5. Summary

We have presented a description and examples of a problem where one needs to identify places where invariants are required in a UML class diagram. This is due to the inherent weakness of graphical notations. The basic identification technique is to identify circularities created by the relationships between classes; the two most common being the triangle and a square. Other “polygons” do exist but are less important and contribute less to the overall information context and may be already expressed by or can be mapped into smaller circularities and their invariants.

We do not specifically address here (and nor was it the aim of this paper to consider) how one detects whether the model is overly constrained [11]. One may even argue that it is better to work backwards from an overly constrained model than to continually patch an underconstrained model.

We have also presented further UML constructs (attributes, composition, aggregation and association classes) which are either mappable into other constructs which makes them amenable to the pattern described here, or, detailed how the language itself allows or denies object sharing.

Implementation of this pattern in a tool has been prototyped (Coral [1]) and utilises simple techniques from graph theory to identify the circularities. Our experiences from this and then mapping using U2B [12] have meant that the specification produced are more amenable to the types of verification seen with theorem proving

References

- [1] Marcus Alanen and Ivan Porres (2004) The Coral Modelling Framework. In: Koskimies, Kai and Kuzniarz, Ludwik and Lilius, Johan and Porres, Ivan (editors) Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004 General Publications, Num: 35, Turku Centre for Computer Science, Jul, 2004
- [2] Nuno Amalio and Susan Stepney and Fiona Polack (2004). Formal Proof from UML Models. In Proceedings: Jim Davies, Wolfram Schulte and Mike Barnett (editors) ICFEM'04, Seattle, USA, 2004. Lecture Notes in Computer Science 3308. Springer 2004.
- [3] Franco Civello (1993). Roles for Composite Objects in Object-Oriented Analysis and Design. In Proceedings of OOPSLA 1993. pp376-393
- [4] Desmond D'Souza and Alan Cameron Wills (1998). Objects, Components and Frameworks with UML - The Catalysis Approach. Addison Wesley. 0-201-31012-0
- [5] Stephan Flake and Wolfgang Mueller (2003). Expressing Property Specification Patterns with OCL. In The 2003 International Conference on Software Engineering Research and Practice (SERP'03), Las Vegas, Nevada, USA. June 2003.
- [6] Jean Memet (ed) (2004) UML-B Specification for Proven Embedded Systems Design. Kluwer Academic Publishers. 1-4020-2866-0
- [7] Bertrand Meyer (1992) Applying Design by Contract. Computer vol 25:10, pp 40.51, October 1992.
- [8] Oliver Ian (2002) Simulation of Software Behaviour using Animation. 16th European Simulation Multiconference ECS02. Darmstadt, Germany, 3-5 June 2002.
- [9] OMG (2004) UML 2.0 Superstructure Specification.
- [10] Mark Richters and Martin Gogolla (2003) Validation of UML and OCL Models by Automatic Snapshot Generation, Sixth International Conference on the Unified Modeling Language - the Language and its applications, 2003
- [11] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan and Mana Taghdiri (2003) Debugging Overconstrained Declarative Models Using Unsatisfiable Cores. John Grundy, John Penix, D., eds., Proceedings of ASE-2003: The 18th IEEE Conference on Automated Software Engineering. IEEE CS Press. November, 2003. Montreal, Canada.
- [12] Colin Snook, Michael Butler and Ian Oliver (2003). Towards a UML Profile for UML-B. University of Southampton Technical Report 8351. October 2003.
- [13] Susan Stepney, Fiona Polack and Ian Toyn (2003) A Z Patterns Catalogue I: specification and refactorings, v0.1 Department of Computer Science, University of York, Technical Report: YCS-2003-349 January 2003.
- [14] Jos Warmer and Anneke Kleppe (2003) The Object Constraint Language 2nd Edition. Addison Wesley. 0-321-17936-6
- [15] Kirk Wilson and Ian Maung (2002) UML Patterns for Modelling Inference Rules Using OCL. Available at: neptune.irit.fr/Biblio/02-12-15.pdf