hal-00182180, version 1 - 30 Oct 2007

Towards an incremental development of UML specifications

Boulbaba Ben Ammar[1,2], Mohamed Tahar Bhiri[2] and Jeanine Souquières[1]

[1] LORIA - Nancy University
615 rue du Jardin Botanique
F-54602 Villers-lès-Nancy, France
{Boulbaba.Ben-Ammar, Jeanine.Souquières}@loria.fr

[2] MIRACL Laboratory - Faculté des Sciences de Sfax
B. P. 802 - 3018, Sfax - Tunisia
tahar_bhiri@yahoo.fr

Abstract. Specifying complex systems is a difficult task which cannot be done in one step. Step-by-step development processes have been studied using formal methods, based on refinement mechanisms. The refinement is a key feature for incrementally developing more and more detailed models, preserving correctness in each step. Our purpose is to instantiate this process when using UML/OCL notations. We illustrate it by some development steps of an access control system. At each step, decisions are formalized in terms of different UML notations, making evolve an initial model which expresses the fundamental properties of the system. We show that these properties are preserved by each development step.

Key words: Step-by-step development process, UML, OCL, refinement, verification

1 Introduction

Specifying a complex system is a difficult task which cannot be done in one step. Step-by-step development processes have been largely studied in formal approaches, based on refinement mechanisms. Usually, two kinds of refinement are highlighted, horizontal refinements and vertical refinements. The horizontal refinement consists in introducing new viewpoints and new details in an existing model. Each introduction of details to a model leads to a new model which must be coherent with the previous one. If it is the case, we said that the second model refines the first one. The vertical refinement consists in going from an abstract model to a more concrete one, for example by reducing the indeterminism or by strengthening the guard or the postcondition of operations. The concrete model is a realization of the abstract model.

The concept of refinement is an important aspect in formal methods, such as the B method [1] or Object Perfect Developer [6], with the verification of the

hal-00182180, version 1 - 30 Oct 2007

correctness: a refinement is correct if the properties of the abstract model are verified by the concrete model. This notion of refinement between two models, an abstract and a more concrete one, is formally defined and can be mathematically proven by support tools. Meanwhile, there is a lack of a methodological studies related to the incremental development of complex system using the refinement mechanisms [8].

An incremental development is natural and relevant. Nevertheless it is currently difficult to use it with UML notations [12, 18]. Indeed, a rigorous definition of the refinement concept doesn't exist in such notations.

The concept of refinement has been highlighted in several studies. In [19], rules are proposed to check the behavioral consistency between the behavior of objects life cycles. The specialization is defined as a refinement incremented by an extension. Claudia Pons and all. [13, 17, 14, 15] have developed a tool called E-Platero [16] which supports some refinement development of class diagrams. The E-Platero tool proposes a relation of refinement equivalent to the one proposed in Object-Z [7]. Shen and Low [9] propose refinement rules for the generalization and association relationships. These rules are described on the meta-model as stereotypes. In [5], the authors define a formal notion of refinement through unification, explaining the concept of correspondence between specifications, known as conformance relationships.

The goal of this paper is to present an incremental development of an UML specification. We start with an abstract initial model. For each refinement step, we present the taken decisions and how we express them in terms of OCL notation [11], UML class diagram and invariant on the global system. This development case study is widely inspired by Abrial [2].

In section 2, we present some development steps of the access control case study using the UML/OCL notations. In section 3, we present lessons learn from this development case study and how the refinement steps can be supported by refinement patterns. Section 4 concludes this paper and proposes different perspectives.

2 Access control case study: some development steps

The access control case study is in charge to control the access of authorized persons to different buildings [3]. The authorization should allow a person, controlled by the system, to enter into certain buildings, and not into others. The exit of a person of a building is also controlled by the system, in order to know at any moment who is inside a given building.

2.1 Initial specification

The main goal of the system is to control authorized persons to enter and leave the buildings and to manage the dynamic situation of the presence of persons in the buildings.

UML class diagram. A first class diagram corresponding to this description is presented in Fig. 1 in which:

1. the two classes, *Person* and *Building* represent the set of persons and the set of buildings,
2. the two associations represent the authorization and the situation relationship between *Person* and *Building*. The multiplicity of the association *situation* expresses the fact that, at any moment, a person must be in at least one building. The parameter *unique* for the *authorization* association expresses the fact that this association cannot be modified.

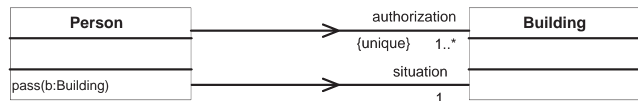

Fig. 1. Initial model

System properties. The authorizations can be defined as a set of possible couples (person, building), where each couple links a person to a building in which he is authorized to enter. This set of couples is expressed as a binary relation between *Person* and *Building* and formalized in OCL as presented in (1).

```
Context Person
--the binairy relation is expressed by two nested iterations
def :aut :Set(TupleType(p :Person, b :Building))=
    Person ::allInstances→iterate(pr :Person ;
      resultset :Set(TupleType(p :Person, b :Building))=Set{}|
    pr.authorization→iterate(bt:Building;|resultset.including(tuple{pr,bt})))
```
(1)

Moreover, the system has to manage the dynamic situation of the persons inside the buildings. We introduce a new set, *sit*, linking each person to the building in which he is located (2):

```
Context Person
--sit is a total function, expressed by one iteration
def :sit :Set(TupleType(p :Person, b :Building))=
    Person ::allInstances→iterate(pr :Person ;
      resultset :Set(TypleType(p :Person, b :Building))=Set{}|
    resultset.including(tuple{pr,pr.situation}))
```
(2)

Some important properties of the model are to be added:

- the set of persons is not empty,

hal-00182180, version 1 - 30 Oct 2007

- each person is authorized to enter into given buildings and not others,
- the authorization is a permanent assignment,
- at any time, each person can only be in one building,
- any person in a given building is authorized to be there.

These properties can be formalized as shown in (3).

```
Context Person
inv: Person ::allInstances→notEmpty()
    and self.authorization→size()>= 1
    and self.authorization→forAll(b|self.authorization→isUnique(b))
    and self.situation→size()= 1
    and aut→includingAll(sit)
```
(3)

The property, the set of buildings is not empty, is expressed in (4).

```
Context Building
inv: Building ::allInstances→notEmpty()
```
(4)

Method definition. At this level of abstraction, we can observe the method *pass* of the class *Person* which allows the entry of a person into a building. This event should be able to occur only if the person is authorized to be in the given building and if it is not already there (5).

```
Context Person :: pass(b : Building)
pre: aut→includes(tuple{self,b})
    and sit→excludes(tuple{self, b})
post: sit→including(tuple{self, b})
```
(5)

Conclusion. The invariant of the system is obtained by the conjunction of the invariant of each class of the class diagram, i.e. the invariant of *Person* (3) and *Building* (4). So, it is easy to prove that the unique observable method, *pass*, maintains these properties.

2.2 First refinement

A person can go from a building in which he is to another where he is authorized to enter if these two buildings are connected, i.e. the first one communicates with the second.

UML class diagram. This communication between buildings is expressed by a navigable association named *communication* which links the buildings (see Fig. 2).

hal-00182180, version 1 - 30 Oct 2007

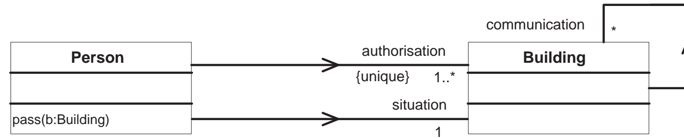

Fig. 2. First refinement

System properties. The concept of communication between buildings is defined as a binary relation between buildings, *com*, as presented in (6).

```
Context Building
−−com describes a binary relation
def :com :Set(TupleType(b1 :Building, b2 :Building))=
    Building ::allInstances→iterate(b1 :Building ;
      resultset :Set(TupleType(b1 :Building, b2 :Building))=Set{}|
    b1.communication→iterate(b2:Building ;|
      resultset.including(tuple{b1,b2})))
```
(6)

A constraint stipulating that two buildings which communicate between each other must be necessarily distinct, has to be introduced (7).

```
Context Building
inv: −−in addition to the invariant (4)
    and self.communication→excludes(self)
```
(7)

Method definition. To satisfy this first refinement, we have to strengthen the guard of the method *pass* as presented in (8).

```
Context Person :: pass(b : Building)
pre: aut→includes(tuple{self, b})
    and self.situation.communication→includes(b)
post: sit→including(tuple{self, b})
```
(8)

Safety properties. In the current state of the model, if a person is in the building *b* and has no authorization to be in any of the buildings which communicates with *b*, this person is blocked in *b*. Therefore, it is necessary to add a supplementary requirement saying that no person must remain blocked in a building.

This implies that each person *p* authorized to enter into the building *b* is also authorized to go at least into another building *c* which communicates with *b*. Thus, we define a new set of couples (p, b), *autSeqcomInv* (9), where *p* is authorized to enter into *c* (*aut→includes(tuple{p, c})*), such that *b* communicates with *c* (*com→includes(tuple{b, c})*). This relation is defined as a sequential composition of the relation *aut* and the inverse of the relation *com*, denoted *comInv*

hal-00182180, version 1 - 30 Oct 2007

(11). This relation is defined by the cartesian product of *Person* and *Building*, denoted *personCartbuilding* (10).

```
Context Person
−−sequential composition of aut and comInv
def :autSeqcomInv :Set(TupleType(p :Person, b :Building))=
      personCartbuilding→iterate(tuple(x :Person, z :Building) ;
        resultset :Set(TupleType(p :Person, b :Building))=Set{}|
      if Building ::allInstances→exists(y :Building|aut→includes(tuple{x,y})
        and Building ::comInv→includes(tuple{y,z}))
      then resultset.including(tuple{x,z}) endif)
```
(9)

```
Context Person
−−cartesian product of Person and Building
def :personCartbuilding :Set(TupleType(p :Person, b :Building))=
      Person ::allInstances→iterate(pr :Person ;
        resultset :Set(TupleType(p :Person, b :Building))=Set{}|
      Building ::allInstances→iterate(bt :Building ;|
        resultset.including(tuple{pr,bt})))
```
(10)

```
Context Building
−−relation inverse of com
def :comInv :Set(TupleType(b1 :Building,b2 :Building))=
      Building ::allInstances→iterate(bt1 :Building ;
        resultset :TupleType(b1 :Building, b2 :Building)=Set{}|
      bt1.communication→iterate(bt2 :Building ;|
        resultset.including(tuple{bt2,bt1})))
```
(11)

A new invariant has to be introduced in the class *Person*:

```
Context Person
inv: −−in addition to the invariant (3)
      and autSeqcomInv→includesAll(aut)
```
(12)

Conclusion. The second definition of the method *pass* given in (8) refines the definition (5), because the action is identical in both cases and the guard of the second: *aut→includes(tuple{self, b})* ***and*** *self.situation.communication→ includes(b)*, is clearly stronger than the first one: *aut→includes(tuple{self,b})* ***and*** *sit→excludes(tuple{self, b})*.

In other words, if a person is in a building which communicates with *b*, then the building where this person finds himself is certainly different from *b* since *b* cannot communicate with itself; therefore the first condition holds.

2.3 Second refinement

A person can go from one building to another one via one-way doors. This leads us to introduce the origin and destination buildings for each door. Each door is equipped with two lights, a green one indicating that the person can enter the building and a red light indicating that he cannot enter.

hal-00182180, version 1 - 30 Oct 2007

hal-00182180, version 1 - 30 Oct 2007

UML class diagram. It is modified as follows:

1. introduction of the class *Door* which represents the set of doors, with a set of methods,
2. introduction of two classes *GreenLight* and *RedLight* which represent the set of the green and red lights. As both are a component of the class *Door*, we introduce a composition relationship between them and the class *Door*,
3. introduction of two associations, *origin* and *destination*,
4. introduction of the association *accepted* which represents the person accepted at a door,
5. removal of *communication*, the recursive association among buildings. This association has been refined by the introduction of the one-way doors modeled by the class *Door* and the two associations *origin* and *destination*.

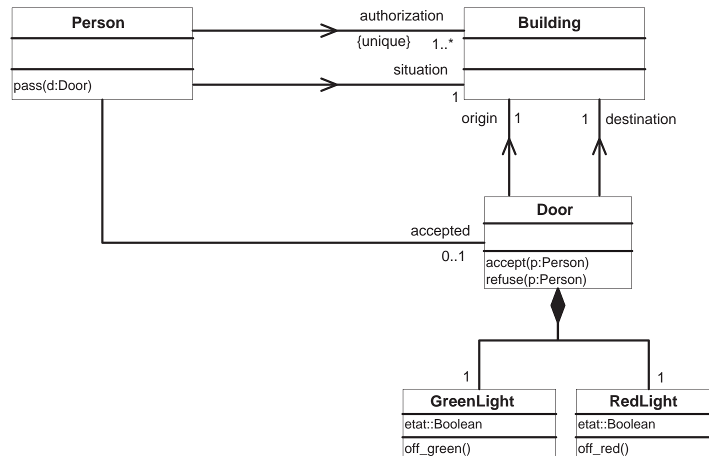

Fig. 3. Second refinement

System properties. We define two sets, *org* (door, origin-building) and *dst* (door, destination-building), formalized in (13) and (14).

```
Context Door
−−org describes a total function
def : org: Set(TupleType(d : Door, b : Building))=
      Door ::allInstances→iterate(dr :Door ;
        resultset :Set(TupleType(d : Door, b : Building))=Set{}|
      resultset.including(tuple{dr,dr.origin}))
```
(13)

```
Context Door
−−dst describes a total function
def : dst: Set(TupleType(d : Door, b : Building))=
      Door ::allInstances→iterate(dr :Door ;
        resultset :Set(TupleType(d : Door, b : Building))=Set{}|
      resultset.including(tuple{dr,dr.destination}))
```
(14)

Doors can be physically blocked. A person can only get through a door if it is accepted and a door can only be accepted for one person at a time. We introduce a new set, *dap*, of (person, door) corresponding to the set of pairs linking a person to the door which is accepted for him, as shown in (15).

```
Context Person
−−dap describes a partial injection
def : dap: Set(TupleType(p : Person, d : Door))=
      Person ::allInstances→iterate(pr :Person ;
        resultset :Set(TupleType(p : Person, d : Door))=Set{}|
      if pr.accepted→size()=1 then
        resultset.including(tuple{pr,pr.accepted}) endif)
```
(15)

The removal of the association *communication* requires a revision of the definition of the set *com*, by adding a new invariant to the class *Building* (16): for all the doors, the buildings of origin and destination represent exactly the pairs of buildings implied in the relation *com*. This definition is called a gluing invariant in formal methods.

```
Context Building
inv: −−in addition to the invariant (7)
      and com = orgInvSeqdst
```
(16)

orgInvSeqdst(17) is defined as a sequential composition of two functions: *orgInv* "the inverse function of *org*" (19) and *dst*. This relation is defined by the cartesian product of *Building* and *Building* and is denoted by *buildingCartbuilding* (18).

```
Context Building
−−sequential composition of orgInv and dst
def :orgInvSeqdst :Set(TupleType(b1 :Building, b2 :Building))=
      buildingCartbuilding→iterate(tuple(x :Building, z :Building) ;
        resultset :Set(TupleType(b1 :Building, b2 :Building))=Set{}|
      if Door ::allInstances→exists(y :Building|orgInv→includes(tuple{x,y})
        and Building ::dst→includes(tuple{y,z}))
      then resultset.including(tuple{x,z}) endif)
```
(17)

```
Context Building
−−cartesien product of buildings
def :buildingCartbuilding :Set(TupleType(b1 :Building, b2 :Building))=
      Building ::allInstances→iterate(b1 :Building ;
        resultset :Set(TupleType(b1 :Building, b2 :Building))=Set{}|
      Building ::allInstances→iterate(b2 :Building ;|
        resultset.including(tuple{b1,b2})))
```
(18)

hal-00182180, version 1 - 30 Oct 2007

```
Context Building
−−inverse function of org
def :orgInv :Set(TupleType(b :Building,d :Door))=
      Door ::allInstances→iterate(dr :Door ;
        resultset :TupleType(b :Building,d :Door)=Set{}|
        resultset.including(tuple{dr.origin,dr})))
```
(19)

The property, an origin-building and a destination-building is associated to each door, has to be added to the class *Door*, formalized as presented in (20).

```
Context Door
inv: Door.allInstances→notEmpty()
      and self.origin→size()= 1
      and self.destination→size()= 1
      and self.greenLight→size()= 1
      and self.redLight→size()= 1
```
(20)

A new invariant is added in the class *Person* (21), making clear the condition for acceptance to a door for a given person. It is defined in terms of two sets, *dapSeqorg*, for a door to accept a person, this person should be inside the building of origin of that door, and *dapSeqdst*, a person should be authorized to enter the destination building of that door. These two sets are defined as a sequential composition of *dap* and *org* for the first one and *dap* and *dst* for the second one, and not presented here.

```
Context Person
inv: −−in addition to the invariant (12)
      and aut → includesAll(dapSeqdst)
      and sit → includesAll(dapSeqorg)
```
(21)

Next properties are related to the lights:

- the green light of a door is lit when the person is accepted,
- the red and green lights of a same door cannot be lit simultaneously

and are formalized by the introduction of a new invariant in the class *Person* (22) and in the class *Door* (23).

```
Context Person
inv: −−in addition to the invariant (21)
      and accepted.notEmpty() implies accepted.greenLight.etat
```
(22)

```
Context Door
inv: −−in addition to the invariant (20)
      and ((greenLight.etat implies not(redLight.etat))
        or (redLight.etat implies not(greenLight.etat)))
```
(23)

hal-00182180, version 1 - 30 Oct 2007

Method definitions. The predicate *admitted*, which has a boolean result expresses the condition of admission of a person. This predicate has to be defined in the class *Door* as shown in (26). Such formalization needs the definition of the domain and the codomain of the partial injection function *dap*.

```
Context Person
−−domain of dap partial function
def :dom :Set(Person)=
      dap→iterate(c:TupleType(p:Person, d:Door);
        resultset:Set(Person)=Set{}|resultset.including(c.p))
```
(24)

```
Context Person
−−codomain of dap partial function
def :ran :Set(Door)=
      dap→iterate(c:TupleType(p:Person, d:Door);
        resultset:Set(Door)=Set{}|resultset.including(c.d))
```
(25)

```
Context Door
def :admitted(p : Person): Boolean =
      (self.origin = p.situation)
      (Person.aut→includes(tuple{p, self.destination}))
      (Person.dom→excludes(p))
```
(26)

In the same way, the methods *accept* and *refuse*, which correspond to the discovery of a person wishing to go from one building to another, accept and reject accordingly. The methods *off_green* and *off_red*, which are presented in the class *GreenLight* and *RedLight*, change the value of the attribute *etat* to *False*.

The method *pass* has to be refined: it can be triggered for a door which green light is on. The person is allowed to pass and the method results in turning off the green light.

```
Context Person :: pass(d : Door)
pre: ran→includes(d)
post: (sit→including(d.person, d.destination)
      and ran→excludes(d)
```
(27)

Conclusion. The method *pass* is refined by strengthening its precondition and postconditions. Each new method is considered as a refinement of an operation which does nothing.

3 Lessons learned

A step by step development, based on refinement mechanisms, presents the following advantages:

- it introduces progressively details and choices,

hal-00182180, version 1 - 30 Oct 2007

- it strongly motivates the introduction of the different ingredients in the model.

The formalization of important properties of the system and their refinement gives a better comprehension of the system. It allows verifications to be done through out the life cycle of development such as:

1. preservation of the invariant of the system, i.e., each operation must preserve the invariant.
2. proof of the refinement at each refinement step, i.e., we can verify that the concrete model satisfies the properties of the previous model.

The verifications carried out by our incremental development are similar to the proof obligations generated by the tools of formal methods. Therefore, the refinement concept defined in the formal methods could be used to develop a UML specification, using OCL constraints to ensure the trustworthy refinement. This needs to use mathematical notations such as function and relations. It is to be noted that the use of these concepts in OCL is very tedious and demands some assistance for the developers. This approach can be supported by refinement patterns to provide assistance for the developers. In [4], we have defined several refinement patterns for the development of a UML specification. These patterns have been proven by using the B method and could be used as a guide in the development of UML specifications.

4 Conclusion and perspectives

One of the difficult tasks for developers of object oriented software is to find the pertinent classes. Some methodological studies bring several propositions based on different approaches [10]: linguistics, use case, abstract data type and patterns (design and analysis patterns). In these approaches the relevance of the founded classes is related to the competence of the developer. However, this task can be realized easier if the developers employs "divide and rule" strategy. The developers start with an initial abstract model then they proceed by refinement. In each refinement step, the developers should take care of new considerations and try to extract pertinent classes and relationships. The output of this step must be coherent with the previous one. The OCL constraints can be used to ensure the trustworthy application of the refinement. Indeed, the UML language does not allow the expression of all properties and constraints. However the handling of the functions and mathematical relations remains obscure. Therefore it is necessary to extend OCL in order to facilitate its use.

In this paper, we present a case study showing some guidelines to refine a UML specification. We start from first general model which expresses in an abstract way the property of the system. Each refinement step subsequently is guided by the evaluation of the model, by adding new details. A verification of the preservation of the invariant also ensures the verification of the refinement steps. This work is widely inspired by the refinement as defined in formal methods such as B.

hal-00182180, version 1 - 30 Oct 2007

hal-00182180, version 1 - 30 Oct 2007

To verify the preservation of the invariant and operations' refinement, we propose to create an environment which supports both UML and OCL notations. This prototype can verify the reinforcement of the invariant of each class and the pre/post-conditions of each operation by the user. This prototype must be able to support refinement patterns to assist users.

We reflect now on the definitions of the refinement patterns:

– refinement pattern of association relationships: it can be used in the first development step when the *com* association is refined by two new associations *org* and *dst*,
– refinement pattern of inheritance relationships: it can be used after the second refinement for the classes *RedLight* and *GreenLight* which have the same characteristics so it is possible to factorize them in one super class *Light*,
– algorithmic refinement pattern: it can be used before the second refinement when the operation *pass* changes its signature.

A reflection is to preserve the history of the pattern applications in order to have a traceability of the development process. In the requirement case, the developers have the possibility of the back-tracking and reviewing their decisions.

References

1. J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0521496195.
2. J.R. Abrial. Event Driven System Construction. Technical report, ClearSy System Engineering, April 1999.
3. AFADL'2000. Etude de cas : système de contrôle d'accès. In *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels*, 2000. actes LSR/IMAG.
4. B.a Ben-Ammar, M. T. Bhiri, and J. Souquières. Quelques patrons de raffinement pour le développement de diagrammes de classes UML. In *Atelier OCM-SI: 6ème atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information*, 2007.
5. E.A. Boiten and M.C. Bujorianu. Exploring UML refinement through unification. In J. J"urjens, B. Rumpe, R. France, and E.B. Fernandez, editors, *Critical Systems Development with UML - Proceedings of the UML'03 workshop*, number TUM-I0323, pages 47–62. Technische Universitat Munchen, 2003.
6. D. Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. *FM 2003: the 12th International FME Symposium*, September 2003.
7. S. Graeme. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
8. M. Guyomard. Spécification et raffinement en B : deux exemples pédagogiques. *ZB2002 4th International B Conference, Education Session Proceedings*, Janvier 2002.
9. W. L. Low. Using the Metamodel Mechanism to Support Class Refinement. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 421–430. IEEE Computer Society, 2005.

hal-00182180, version 1 - 30 Oct 2007

10. B. Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
11. OMG. *OCL 2.0 Specification, Final Adopted Specification ptc/03-10-14*, October 2003.
12. OMG. *UML 2.0 Superstructure Specification, Final Adopted Specification ptc/03-08-02*, August 2003.
13. C. Pons. On the Definition of UML Refinement Patterns. In *2nd MoDeVa workshop, Model design and Validation*, October 2005.
14. C. Pons. Heuristics on the Definition of UML Refinement Patterns. *Springer-Verlag Berlin Heidelberg*, 2006.
15. C. Pons and D. Garcia. An OCL-Based Technique for Specifying and Verifying Refinement-Oriented Transformations in MDE. In *MoDELS*, pages 646–660, 2006.
16. C. Pons, R. S. Giandini, G. Pérez, P. Pesce, V. Becker, J. Longinotti, and J. Cengia. Pampero: Precise assistant for the modeling process in an environment with refinement orientation. In *UML Satellite Activities*, pages 246–249, 2004.
17. C. Pons, G. A. Perez, R. Giandini, and R. Kutsche. Understanding Refinement and Specialization in the UML. In *2nd International Workshop on MAnaging SPEcialization/Generalization HIerarchies (MASPEGHI 2003)*, October 2003.
18. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1998.
19. M. Schrefl and M. Stumptner. Behavior-consistent specialization of object life cycles. *ACM Trans. Softw. Eng. Methodol.*, 11(1):92–148, 2002.