Heuristics on the Definition of UML Refinement Patterns

Claudia Pons

LIFIA – Computer Science Faculty – University of La Plata and CONICET,
La Plata, Buenos Aires, Argentina
cpons@info.unlp.edu.ar

Abstract. In this article we present a strategy to formalize frequently occurring forms of refinement that take place in UML model construction. Such strategy consists in recognizing a set of well founded refinement structures in a formal language which are then immersed into a UML-based development, giving origin to a set of UML refinement patterns. Apart from providing semi-formal evidence on the presence of refinement structures in object-oriented designs, this strategy made it possible to reveal hidden refinements and to discover weaknesses of the UML language that hinder the specification of refinement. An automatic tool is provided to support model refinement activities.

1 Introduction

Model Driven Development (MDD) [8][16], which prescribes the use of UML [14] as the standard modeling language, aims at introducing techniques for raising the level of abstraction to describe both the problem and its solution, and by clearly establishing methodologies to define the problem and how to move to its solution. The idea promoted by MDD is to use models at different levels of abstraction. A series of transformations are performed starting from a platform independent model with the aim of making the system more platform-specific at each refinement step. However, model transformations are frequently only viewed as a technique for generating models; little is said about guaranteeing the correctness of the generated models. In fact, model transformations should do more than just generate models; in addition, they should generate evidence that the generated models are actually correct. In particular, some of these transformations can be cataloged as refinements in the sense of formal languages [6], thus being amenable to formal verification.

Formal verification of model refinement can be fully exploited only if the language used to create the models is equipped with formal refinement machinery, making it possible to prove that a given model is a refinement of another one, or even to calculate possible refinements from a given model. This refinement machinery is present in most formal specification languages such as Object-Z [6], [21], B [10], and the refinement calculus [2]. Besides, some restricted forms of programming languages can also be formally refined [4]. But, in the standard specification language UML [14], the refinement machinery has not reach a mature state yet. Being UML a language widely used in software development, any effort made towards increasing the robustness of the UML refinement machinery becomes a valuable task which will also contribute to the improvement of MDD. To reach this goal, most researchers

© Springer-Verlag Berlin Heidelberg 2006

have used an “informal-to-formal” approach consisting in translating the graphical notation into a formal language equipped with refinement machinery. For example, the works of Davies and Crichton [5] Engels et al.[7] Astesiano and Reggio [1], Lano and Biccaregui [11], Ledang and Souquieres [12] among others. In this way, UML refinements become formally defined in terms of refinements in the target language. This approach is valuable, and in most cases it allows us to verify and calculate refinements of UML models. However, this approach is insufficient because it does not address the following problems: - *lack of notation to specify refinements (*although the UML Abstraction artifact allows for the explicit documentation of the refinement relationship in UML models, the available features of the Abstraction artifact are frequently insufficient to formally define the relationship); - *presence of hidden refinements:* an important amount of variations of abstraction/refinement remains unspecified, usually hidden under other notations. Those hidden refinements should be discovered and accurately documented [17], [18]; - *missing refinement methodology:* the formalization of the language itself is only the starting point; we also need a stepwise refinement methodology, based on a formal theory, consisting of refinement patterns, rules and guidelines.

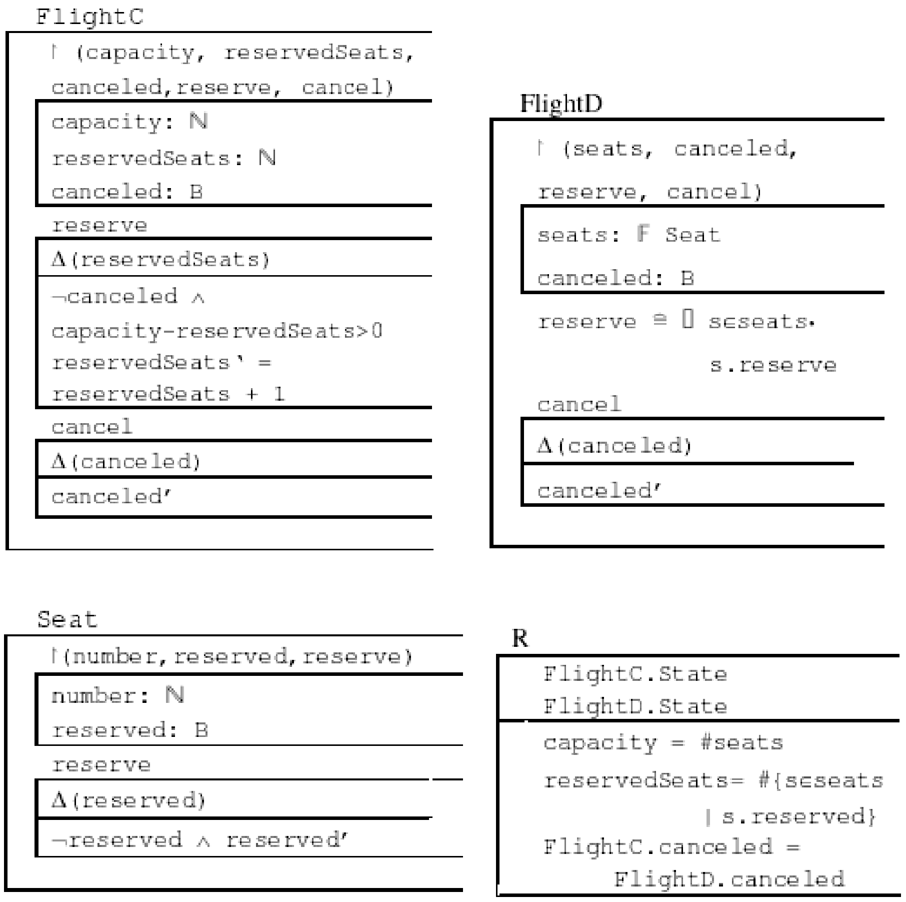
We explored an alternative approach (i.e., a “formal-to-informal” approach) as a complement to the former. According to this approach a formally defined refinement methodology is immersed into a UML-based development. Concretely, well founded refinement structures in the Object-Z formal language provide inspiration to define refinement structures in the UML, which are (intuitively) equivalent to their respective inspiration sources.

The structure of this document is as follows: first, in sections 2 and 3 we describe the results of applying a “formal-to-informal” approach towards the improvement of the UML refinement machinery; we present an extract of a catalog of well-founded Object-Z refinement patterns, each of them giving origin to a list of several UML refinement patterns (each single Object-Z refinement pattern can be analyzed from a number of perspectives, which give rise to a number of UML refinement structures, one for each perspective). Finally, sections 4 discusses related work and conclusions.

2 Object Decomposition Pattern

Description: Composition is a form of abstraction: things are composed of smaller things, and this recursively; the composite represents its components in sufficient detail in all contexts in which the fact of being composed is not relevant and conversely decomposition is a form of refinement: an abstract element is described in more detail by revealing its interacting internal components.

Example: in a flight booking system (figure 1), each flight is abstractly described by its overall capacity and the quantity of reserved seats in its cabin (i.e., class FlightC), then a refinement is produced (i.e.,class FlightD) by specifying in more detail the fact that a flight contains a collection of seats in its interior. In this case seats are described as individual entities whit their own attributes and behavior (a seat has an identification number and a Boolean attribute indicating whether it is reserved or not). In both specifications a Boolean attribute is used to represent the state of the



```
FlightC
↾ (capacity, reservedSeats,
canceled,reserve, cancel)
  capacity: ℕ
  reservedSeats: ℕ
  canceled: B
reserve
  Δ(reservedSeats)
  ¬canceled ∧
  capacity-reservedSeats>0
  reservedSeats' =
  reservedSeats + 1
cancel
  Δ(canceled)
  canceled′

FlightD
↾ (seats, canceled,
reserve, cancel)
  seats: 𝔽 Seat
  canceled: B
reserve ≙ [] s∈seats•
              s.reserve
cancel
  Δ(canceled)
  canceled′

Seat
↾(number,reserved,reserve)
  number: ℕ
  reserved: B
reserve
  Δ(reserved)
  ¬reserved ∧ reserved′

R
  FlightC.State
  FlightD.State
  capacity = #seats
  reservedSeats= #{s∈seats
            | s.reserved}
  FlightC.canceled =
       FlightD.canceled
```

Fig. 1. Refinement induced by Decomposition in Object-Z Classes

flight (open or canceled). The available operations are `reserve` to make a reservation of one seat and `cancel` to cancel the entire flight. The retrieve relation R establishes the connection between both specifications. The refined version of the operation `reserve` selects a seat, ready to be reserved, in a non-deterministic way.

UML Realizations of the Pattern: In this section we describe one UML instantiations of the Object Decomposition Pattern: Object Decomposition in Class Diagrams; other instantiations of the pattern are observed for example in Collaboration and Interaction Diagrams. The OCL language [15], [20] has been used to specify the operation's pre and post conditions. The mapping attached to the abstraction relationship is expressed in an OCL-like language (a discussion on the mapping's language issue is included bellow). Figure 2 shows a refinement of the class FlightC, which was obtained by specifying in detail the fact that a flight contains a collection of seats. The refinement mapping (expressed in pseudo-OCL) states the connection between abstract and refined attributes.

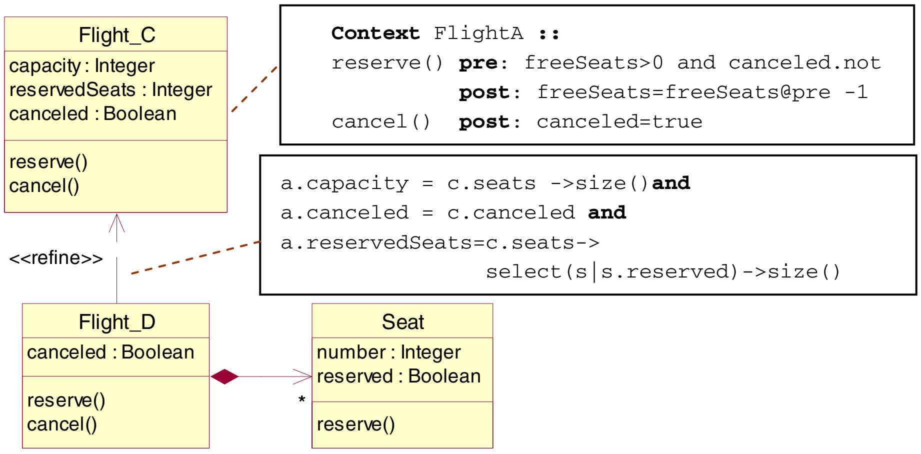

Flight_C
capacity : Integer
reservedSeats : Integer
canceled : Boolean
reserve()
cancel()
Context FlightA ::
reserve() pre: freeSeats>0 and canceled.not
post: freeSeats=freeSeats@pre -1
cancel() post: canceled=true
a.capacity = c.seats ->size() and
a.canceled = c.canceled and
a.reservedSeats=c.seats->
select(s|s.reserved)->size()
<<refine>>
Flight_D
canceled : Boolean
reserve()
cancel()
*
Seat
number : Integer
reserved : Boolean
reserve()

Fig. 2. Refinement induced by Decomposition in UML Class Diagram

Formalization: By applying the definition of downward simulation in Object-Z [6], it is possible to verify the refinement, in the following way:

Initialization:
$\forall$FlightD.State • FlightD.init $\Rightarrow$($\exists$ FlightC.State • FlightC.init $\wedge$ R)

Applicability:
$\forall$FlightC.State;FlightD.State • R $\Rightarrow$ (pre reserveC $\Rightarrow$ pre reserveD)
$\forall$FlightC.State;FlightD.State • R $\Rightarrow$ (pre cancelC $\Rightarrow$ pre cancelD)

Correctness
$\forall$FlightC.State;FlightD.State;FlightD.State'•
R $\wedge$ pre reserveC $\wedge$ reserveD $\Rightarrow$ $\exists$FlightC.State'• R' $\wedge$ reserveC
$\forall$FlightC.State;FlightD.State;FlightD.State'•
R $\wedge$ pre cancelC $\wedge$ cancelD $\Rightarrow$ $\exists$FlightC.State'• R'$\wedge$ cancelC

Discussion:
Issues on hidden refinement: In UML, decomposition is not considered as a form of model refinement. This pattern reveals a particular case of hidden refinement: UML models with composite association implicitly specify refinement relationship. See [18] for a detailed discussion on this issue.

Issues on the specification of delegation: The behavior of the class FlightC was specified in figure 2 as follows:

```
Context FlightC :: reserve()
pre: capacity-reservedSeats>0 and not canceled
post: reservedSeats=reservedSeats@pre + 1
```

In general, the structural decomposition of an object is accompanied by a behavioral decomposition realized through delegations. In the abstract specification it seems that the object carries out its tasks by itself, but in the refined version we can observe that the object delegates sub-tasks to its constituent objects. Let us present the OCL specification of the constituent class Seat:

```
Context Seat :: reserve()
pre:  not reserved
post: reserved
```

To specify the behavior of the refined class FlightD we need to write an OCL expression that is (intuitively) equivalent to the simple following Z expression, which makes a non-deterministic choice of a seat to be reserved:

```
reserve ≙ □ s ∈ seats • s.reserve
```

The most approximated OCL expression we obtain is:

```
Context FlightD :: reserve()
pre:  seats -> select (s| not s.reserved) -> notEmpty()
post: let s=seats->any(s| not s.reserved) in s^reserve()
```

In this pattern we face the OCL restriction that non query operations, such as the `reserve()` operation, are not allowed to be referred to within OCL expressions. Without this facility the specification of delegation in OCL is only possible through the use of OCL Message expressions, allowing us to express messages sent between objects through the hasSent operator ^ [17, pg.29-31]. These expressions are little appropriate for building specifications because they talk about explicit communication between objects instead of describing the effects of the communication in a declarative form. The expression `s^reserve()` in the specification of operation `FlightD::reserve()` evaluates `true` if a `reserve()` message was sent to `s` during the execution of the operation. Moreover, the fact that a method has been called during the execution of an operation, does not assure that its effects were accomplished. The only thing we can assure is that sometime during the execution of `FlightD::reserve()`, the operation `reserve()` has been called over the `Seat` instance `s`. Furthermore, to specify that the operation has already returned we should use the OCL operation `hasReturned()`, however this introduces annoying complication on the specification.

Issues on the syntax to specify the retrieve relation: Graphically, the abstraction mapping describing the relation between the attributes in the abstract element and the attributes in the concrete element is attached to the refinement relationship; however, OCL expressions can only be written in the context of a Classifier, but not of a Relationship. Then, if we want to use the OCL to express the abstraction mapping we need to determine which the context of the expression is. On the Z side, the context of the abstraction mapping is the combination of the abstract and the concrete states; however, a combination of Classifiers is not an OCL legal context; consequently we might write the mapping in the context of the abstract (or the concrete) classifier only, in the following way:

```
Context a:FlightC
def: mapping(c : FlightD) : Boolean =
  a.capacity = c.seats ->size() and a.canceled = c.canceled
  and a.reservedSeats=c.seats ->select(s|s.reserved)->size()
```

The transformation from the pseudo-OCL expressions in figures 2 to their corresponding legal OCL expressions above can be generically defined in the following way: let d be a refine relationship with meta-attributes d.supplier (the abstract classifier), d.client (the concrete classifier) and d.mapping.body (the pseudo OCL expression specifying the mapping). We derive a Boolean operation definition in the context of the abstract classifier:

```
Context a: anAbstractElement
def:mapping(c:aConcreteElement):Boolean=aBoolOclExpression
```

Where anAbstractElement, aConcreteElement and aBoolOclExpression are replaced by d.supplier.name, d.client.name and d.mapping.body respectively.

Issues on the verification process: Verification heuristics can be defined for this refinement pattern. On the one hand, to verify the refinement conditions we can translate the UML diagram back to Object-Z using already developed strategies such as the one proposed by Kim and Carrington in [9]. Then, verification is carried out on the formal specification. Alternatively, we might remain on the UML+OCL side by defining refinement conditions in OCL in a similar style to the Object-Z refinement conditions [6].

3 Non-atomic Operation Refinement Pattern

Description: In the refinements we have analyzed so far the abstract and concrete classes have been conformal, i.e., here has been a 1-1 correspondence between the abstract and concrete operations. Conformity can be relaxed allowing the abstract and concrete specifications to have different sets of observable operations. This case takes place when the abstract operation is refined not by one, but by a combination of concrete operations, thus allowing a change of granularity in the specification.

Example: the flight booking system specified in the schema BookingSystemD in figure 3 records a sequence of flights which can be reserved through the system; then the schema BookingSystemE defines a refinement of operation `reservation` into *checkPassenger ⨾checkFlight recordReservation.*

UML Realizations of the Pattern: This section contains the description of one of the instantiation of the Non-Atomic operation Refinement Pattern - non-atomic operation refinement in class diagrams. This pattern can also be instantiated in Use Case, Interaction and Activity diagrams, among others. Figure 4 contains an example of non-atomic operation refinement in a class diagram; the refinement relationship specifies that the abstract class BookingSystemD has been refined by the more concrete class BookingSystemE; in particular, the abstraction mapping states that operation reservation() has been refined by the combination of three concrete operations.

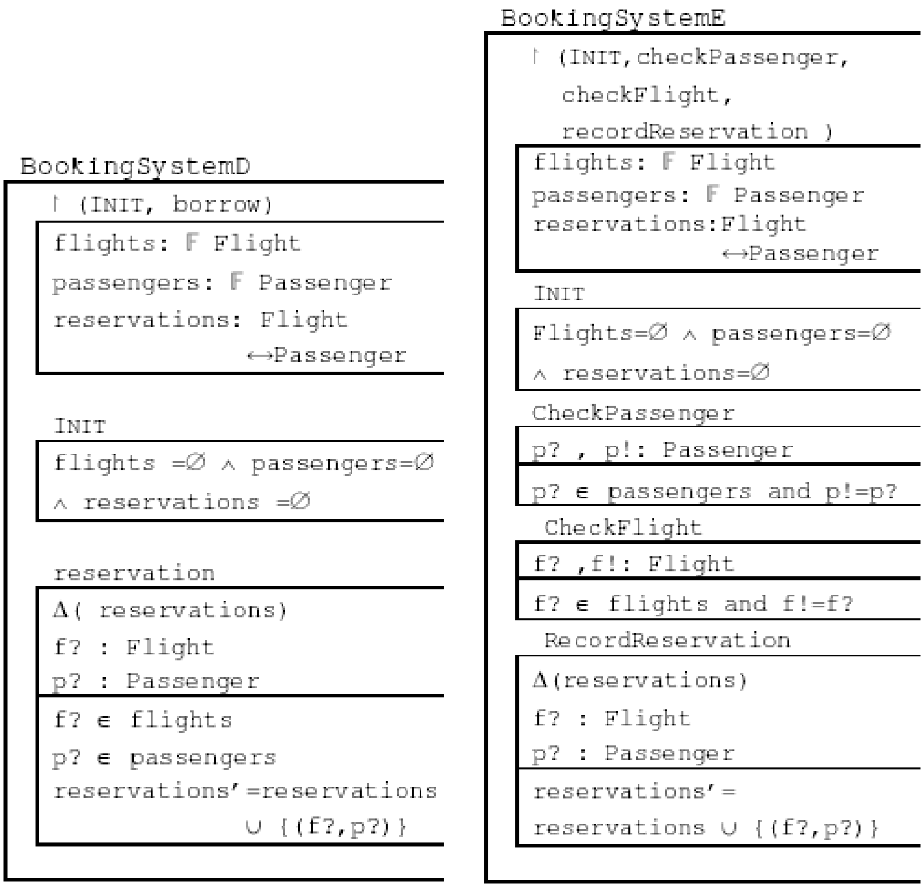


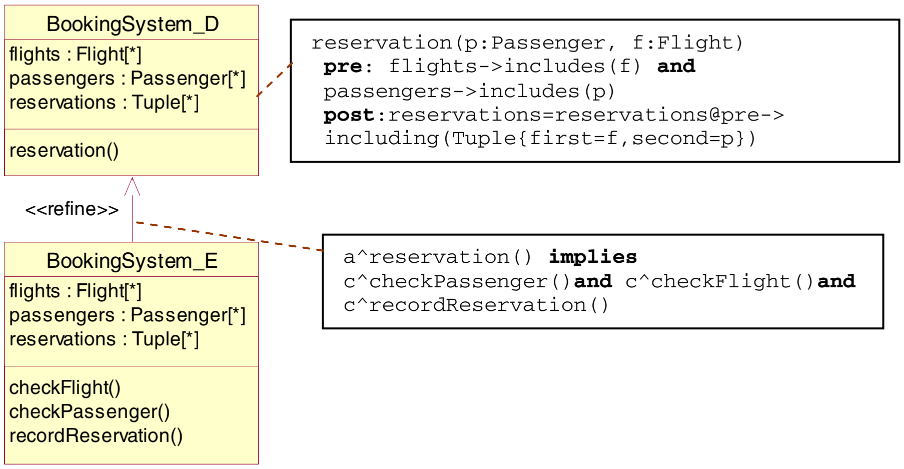
Fig. 3. Non-atomic Operation Refinement in Object_Z Classes

Discussion:

Issues on the syntax to specify the retrieve relation: It was already discussed in the definition of previous patterns, that although in the diagram the mapping specifying the relation between the abstract operation `reservation` () and its refinement is attached to the refinement relationship, the mapping should be actually defined in the context of some of the involved classes, as follows:

```
Context a: BookingSystemD
def: mapping(c : BookingSystemE) : Boolean =
     c^checkPassenger()and c^checkFlight()and
     c^recordReservation() implies a^reservation()
```

Issues on the syntax to specify composition of behaviors: It is possible to express that `reservation`() is realized as the combination of the three operations, however message expressions do not provide the way to specify execution order. The fact that the reservation should be checked before being recorded cannot be expressed.



```
reservation(p:Passenger, f:Flight)
 pre: flights->includes(f) and
 passengers->includes(p)
 post:reservations=reservations@pre->
 including(Tuple{first=f,second=p})
```

```
a^reservation() implies
c^checkPassenger()and c^checkFlight()and
c^recordReservation()
```

Fig. 4. Non-atomic Operation Refinement in UML Class Diagrams

Although we agree that other UML artifacts (such as Interactions) can be used to specify this concern, we believe that OCL suffers from the lack of an operation calculus (like the one of Z) allowing us to specify sequential and parallel composition of operations. Besides, the operational semantics of the OCL hasSent operation (^) given in [15] does not fit the intended semantics of a refinement mapping which declares the equivalence of both behaviors without talking about the actual execution of them.

4 Conclusion

The aim of this work is not to formalize UML refinements in Object-Z, but to substantiate a number of intuitions about the nature of possible refinement relations in UML, and even to discover particular refinement structures that designers do not perceive as refinements in UML. Focusing on the refinement structures of Object-Z we obtained a compact catalog of refinement patterns that can be applied during the UML modeling process; each graphical refinement pattern being based on a formal refinement pattern.

Similar proposal were presented in [3], where Boiten and Bujorianu explore refinement indirectly through unification; the formalization is used to discover and describe intuitive properties on the UML refinements. On the other hand, Liu, Jifeng, Li and Chen in [13] use a formal specification language to formalize and combine UML models. Then, they define a set of refinement laws of UML models to capture the essential nature, principles and patterns of object-oriented design, which are consistent with the refinement definition.

The strategy we propose in this article apart from providing formal evidence on the presence of refinement structures in object-oriented designs made it possible to reveal hidden refinements and to discover weaknesses of the UML language that prevent designers from specifying frequently occurring forms of refinement. Besides, the understanding of refinement patterns is more precise, since each pattern is described from both an intuitive and a mathematical point of view.

Finally, the overall contribution of this research is to clarify the abstraction/refinement relationship in UML models, providing basis for tools supporting the refinement driven modeling process. In this direction we are building ePLATERO [19] that is a plug-in to the Eclipse development environment, based on the heuristics that have been proposed in this article. ePlatero will assist a variety of activities related to refinement, such as explicit documentation, semi-automatic discovering of hidden refinements, refinement-step checking, constraint refinement and refinement patterns application.

References

1. Astesiano, E., and, Reggio, G.: An Algebraic Proposal for Handling UML Consistency. Workshop on Consistency Problems in UML-based Software Development,. Blekinge Institute of Technology Research Report (2003)
2. Back, R. and, von Wright, J.: Refinement Calculus: a Systematic Introduction, Graduate Texts in Computer Science, Springer Verlag. (1998)
3. Boiten, E.A., and Bujorianu, M.C.: Exploring UML Refinement through Unification. Proceedings of the UML'03 Workshop on Critical Systems Development with UML, J. Jurjens, B. Rumpe, et al., (eds) -TUM-I0323, Technische Universitat Munchen, September 2003
4. Cavalcanti, A., and Naumann, D.: Simulation and Class Refinement for Java. In Proceedings of ECOOP 2000 Workshop on Formal Techniques for Java Programs (2000)
5. Davies, J., and Crichton, C.: Concurrency and Refinement in the Unified Modeling Language. Electronic Notes in Theoretical Computer Science, Elsevier **70** 3 (2002)
6. Derrick, J., and Boiten, E.: Refinement in Z and Object-Z. Foundation and Advanced Applications. FACIT, Springer (2001)
7. Engels, G., Küster, J., Heckel, R., and Groenewegen L.: A Methodology for Specifying and Analyzing Consistency of Object Oriented Behavioral Models. Procs. of the IEEE International Conference on Foundation of Software Engineering. Vienna (2001)
8. Kent, S.: Model Driven Engineering. Integrated Formal Methods: Third International Conference, Turku, Finland, May 15-17, 2002. LNCS **2335**, Springer-Verlag (2003)
9. Kim, S., and Carrington, D.: Formalizing the UML Class Diagrams using Object-Z, Proceedings UML´99 Conference, Lecture Notes in Computer Sciencie **1723** (1999)
10. Lano, K.: The B Language and Method. FACIT. Springer (1996)
11. Lano, K., Biccaregui, J.: Formalizing the UML in Structured Temporal Theories, 2nd. ECOOP Wrk. on Precise Behavioral Semantics, TUM-I9813, Technische U. Munchen (1998)
12. Ledang, H., and Souquieres, J.: Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. Proceedings of Asia-Pacific SE Conference 2002. IEEE Computer Society. Australia. December 4-6, 2002
13. Liu, Z., Jifeng H., Li, X. Chen Y.: Consistency and Refinement of UML Models. Third International Workshop, Consistency Problems in UML-based Software Development III. Satellite event of <<UML>> 2004. Lisbon, Portugal, October 11, 2004
14. UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification. August 2003. http://www.omg.org.

15. OCL 2.0. OMG Final Adopted Specification. October 2003
16. Object Management Group, *MDA Guide*, v1.0.1, omg/03-06-01, June 2003
17. Pons, C., Pérez, G., Giandini, R., Kutsche, and Ralf-D.: Understanding Refinement and Specialization in the UML. 2nd International Workshop on MAnaging SPEcialization/Generalization Hierarchies (MASPEGHI). In IEEE ASE 2003, Canada (2003)
18. Pons, C. and Kutsche, R-D.: Traceability Across Refinement Steps in UML Modeling. Workshop in Software Model Engineering, 7th International Conference on the UML, Lisbon, Portugal. October 11, 2004.
19. Pons C., Giandini R., Pérez G., Pesce P., Becker V., Longinotti J., and Cengia J.: Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. In UML Modeling Languages and Applications: UML 2004 Satellite Activities, Revised Selected Papers. Lecture Notes in Computer Science **3297**, Springer, Oct., 2004. The tool can be downloaded from http://sol.info.unlp.edu.ar/eclipse
20. Richters, M., and Gogolla M.: OCL-Syntax, Semantics and Tools. in Advances in Object Modelling with the OCL. Lecture Notes in Computer Science **2263**, Springer (2001)
21. Smith, G.: The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers. ISBN 0-7923-8684-1. (2000)