Practical Verification Strategy for Refinement Conditions in UML Models

Claudia Pons [1,2] and Diego Garcia [3,1]

[1]LIFIA – Facultad de Informática, Universidad Nacional de La Plata
[2]CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)
[3]UTN (Universidad Tecnológica Nacional)
La Plata, Buenos Aires, Argentina
{cpons,dgarcia}@sol.info.unlp.edu.ar

Abstract. This paper presents an automatic and simple method for creating refinement condition for UML models. Conditions are fully written in OCL, making it unnecessary the application of mathematical languages which are in general hardly accepted to software engineers. Besides, considering that the state space where OCL conditions are evaluated might be too large (or even infinite), the strategy of micromodels is applied in order to reduce the search space. The overall contribution is to propitiate the performing of verification activities during the model-driven development process.

1. Introduction

The stepwise refinement technique facilitates the understanding of complex systems by dealing with the major issues before getting involved in the details. The system under development is first described by a specification at a very high level of abstraction. A series of iterative refinements may then be performed with the aim of producing a specification, consistent with the initial one, in which the behavior is fully specified and all appropriate design decisions have been made.

Stepwise software development can be fully exploited only if the language used to create the specifications is equipped with formal refinement machinery, making it possible to prove that a given specification is a refinement of another specification, or even to calculate possible refinements from a given specification. Robust refinement machinery is present in most formal specification languages such as Object-Z [21], B [11], and the refinement calculus [2], and even in some restricted forms of programming languages [4]. However, the widely-used standard specification language UML [15] lacks for a well-defined notion of refinement.

To alleviate this problem most research on the formalization of UML refinements adhere to the approach of mapping the graphical notation into a formal domain where properties are defined and analyzed. For example the works presented in [1], [5], [7], [10], [12], [13] and [22] among others, belong to this group. They are appropriate to discover and correct inconsistencies and ambiguities of the graphical language, and in most cases they allow us to verify and calculate refinements of (a restricted form of)

Please use the following format when citing this chapter:

Pons, C., García, D., 2006, in IFIP International Federation for Information Processing, Volume 219, Advanced Software Engineering: Expanding the Frontiers of Software Technology, eds. Ochoa, S., Roman, G.-C., (Boston: Springer), pp. 47–61.

UML models. However, such approaches are non-constructive (i.e., they provide no feedback in terms of UML), they require expertise in reading and analyzing formal specifications and generally, properties that should be proved in the formal setting are too complex or even undecidedly.

In [18] and [19] we explored an alternative approach, as a complement to the former; well founded refinement structures in the Object-Z formal language were used to discover refinement structures in the UML, which are (intuitively) equivalent to their corresponding Object-Z inspiration sources. A similar proposal was 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 [14] 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.

In this article we work further on those proposals by enriching such refinement patterns with refinement conditions written in OCL (Object Constraint Language) [16]. The advantage of this approach is that refinement conditions get completely defined in terms of OCL, making it unnecessary the application of languages which are usually hardly accepted by software engineers. OCL is a more familiar language and it has a simpler syntax than Object-Z and other formal languages. Additionally, OCL is part of the UML 2.0 standard and it will probably form part of most modeling tools in the near future.

Furthermore, after defining refinement conditions, the next step is to evaluate such conditions. Ordinary OCL evaluators are unable to determine whether a refinement condition written in OCL holds in a UML model because OCL formulas are evaluated on a particular instance of the model, while refinement conditions need to be validated in all possible instantiations. Therefore, in order to make the evaluation of refinement conditions possible, we extract from the UML model a relatively small number of small instantiations, and check that they satisfy the refinement conditions to be proved. This strategy, called *micromodels of software* was proposed by Daniel Jackson in [9] for evaluating formulas written in Alloy. Later on, Martin Gogolla and colleges in [8] developed a useful adaptation of such technique to verify UML and OCL models. Here we adapt such micromodels strategy to verify refinement conditions.

The structure of this document is as follows: sections 2 serves as a brief introduction to the issue of refinement specification in Object-Z and UML 2.0; section 3 describes the method for creating OCL refinement condition for UML refinement patterns; section 4 explains how the micromodels strategy is applied to verify refinements; finally, the paper closes with a presentation of conclusions and future directions.

2. Refinements Specification in Object-Z and UML

In Object-Z [21], a class is represented as a named box with zero or more generic parameters. The class schema may include local type or constant definitions, at most one state schema and an initial state schema together with zero or more operation schemas. These operations define the behavior of the class by specifying any input and output together with a description of how the state variables change. Operations are defined in terms of two copies of the state: one undecorated copy which represents the before-sate and a primed copy representing the after-state.

For example, figure 1 illustrates the specification of a simple class called Flight, having a state (consisting of two variables) and only one operation.

```
Flight
↾ (freeSeats, reserve)
  freeSeats: ℕ
  canceled: B

  INIT
  freeSeats=300
  canceled=false

  reserve
  Δ(freeSeats)
  freeSeats>0 ∧ ¬canceled
  freeSeats′=freeSeats-1
```

Figure 1: simple Object-Z schema.

Object-Z is equipped with a schema calculus, that is to say a set of operators provided to manipulate Object-Z schemas. The schema calculus makes it possible to create Objects-Z specifications describing properties of other Object-Z specifications. To deal with refinements we need to apply at least the following operators:

- Operator STATE denotes the set of all possible states (i.e., snapshots or bindings) of the system under consideration. For example, Flight.STATE = $\{\langle$freeSeats=x, canceled=t$\rangle \mid 0 \leq x \leq 300 \wedge t \in \{$true, false$\}\}$

- Operator INIT denotes the initial states of a given schema. For example, Flight.INIT = $\{\langle$freeSeats=300 , canceled=false$\rangle \mid \rangle\}$

- Operator pre returns the precondition of an operation schema; that is to say the set of all states where the operation can be applied. For example, pre reserve = $\{\langle$freeSeats = x, canceled=false $\rangle \mid 0 < x \leq 300\}$

- The conjunction of two schemas S and T ($S \wedge T$) results in a schema which includes both S and T (and nothing else).

- Schema implication ($S \Rightarrow T$) denotes the usual logical implication.

In [6] refinement is formally addressed in the context of Object-Z specifications as follows: an Object-Z class C is a refinement (through downward simulation) of the class A if there is a *retrieve relation* R on A.STATE∧C.STATE so that every visible abstract operation A.op is recasted into a visible concrete operation C.op thus the following holds:

(Initialization) $\forall C.\mathrm{STATE} \bullet C.\mathrm{INIT} \Rightarrow (\exists A.\mathrm{STATE} \bullet A.\mathrm{INIT} \wedge R)$

(Applicability) $\forall A.\mathrm{STATE} \bullet \forall C.\mathrm{STATE} \bullet R \Rightarrow (\mathrm{pre}\ A.op \Rightarrow \mathrm{pre}\ C.op)$

(Correctness) $\forall A.\mathrm{STATE} \bullet \forall C.\mathrm{STATE} \bullet \forall C.\mathrm{STATE}' \bullet$
$R \wedge \mathrm{pre}\ A.op \wedge C.op \Rightarrow \exists A.\mathrm{STATE}' \bullet R' \wedge A.op$

This definition allows preconditions to be weakened and non-determinism to be reduced. In particular, applicability requires a concrete operation to be defined wherever the abstract operation was defined, however it also allows the concrete operation to be defined in states for which the precondition of the abstract operation was false. That is, the precondition of the operation can be weakened. Correctness requires that a concrete operation be consistent with the abstract one whenever it is applied in a state where the abstract operation is defined. However, the outcome of the concrete operation only has to be consistent with the abstract, but not identical. Thus if the abstract operation allowed a number of options, the concrete operation is free to use any subset of these choices. In other words, non-determinism can be solved.

On the other hand, the standard modeling language UML [15] provides an artifact named *Abstraction* (a kind of Dependency) with the stereotype <<refine>> to explicitly specify the refinement relationship between UML named model elements. In the UML metamodel an Abstraction is a directed relation from a *client* (or clients) to a *supplier* (or suppliers) stating that the client (the refinement) depends on the supplier (the abstraction). The Abstraction artifact has a meta-attribute called *mapping* designated to record the abstraction/implementation mappings (i.e., the counterpart to the Object-Z *retrieve relation*), which is an explicit documentation of how the properties of an abstract element are mapped to its refined versions, and on the opposite direction, how concrete elements can be simplified to fit an abstract definition. The mapping contains an expression stated in a given language that could be either formal or not. The definition of refinement in the UML standard [15] is formulated using natural language and it remains open to numerous contradictory interpretations.

3. Verification strategy for UML refinement patterns

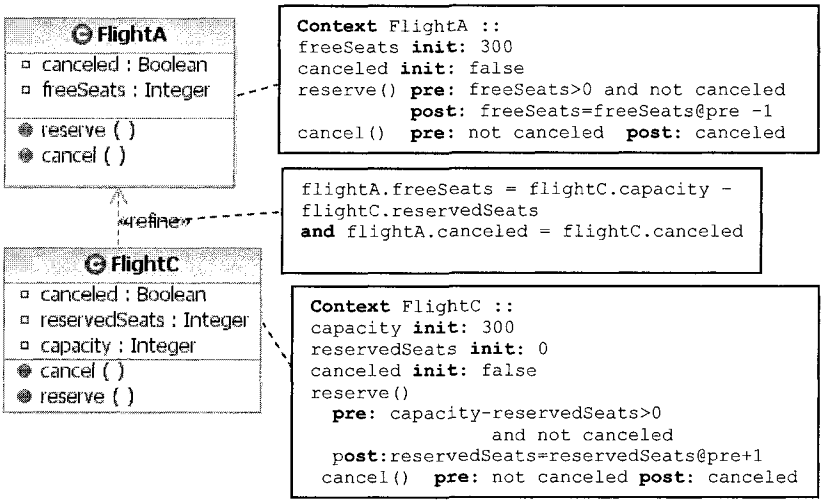
UML refinement patterns [18] [19] document recurring refinement structures in UML models. In this section we describe one of those patterns, the *state refinement pattern*; then we present an algorithm that can be applied on UML models that contain such a pattern in order to automatically create an OCL refinement condition to verify its applicability and correctness. Similar processes were defined to create refinement condition for other patterns in the catalog, but they are not described here due to space limitations.

3.1. The state refinement pattern

A State Refinement takes place when the data structures which were used to represent the objects in the abstract specification are replaced by more concrete or suitable structures; operations are accordingly redefined to preserve the behavior defined in the abstract specification.

An instance of the pattern's structure:

Let M1 be the UML model in figure 2, which is compliant with the structure of the state refinement pattern [19]. M1 contains information about a flight booking system where each flight is abstractly described by the quantity of free seats in its cabin; then a refinement is produced by recording the total capacity of the flight together with the quantity of reserved seats. In both specifications, a Boolean attribute is used to represent the state of the flight (open or canceled). The available operations are `reserve` to make a reservation of one seat and `cancel` to cancel the entire flight. A refinement relationship connects the abstract to the concrete specification. The OCL language [16] [20] has been used to specify initial values, operation's pre and post conditions and the mapping attached to the refinement relationship.



```
Context FlightA ::
freeSeats init: 300
canceled init: false
reserve() pre: freeSeats>0 and not canceled
          post: freeSeats=freeSeats@pre -1
cancel()  pre: not canceled  post: canceled
```

```
flightA.freeSeats = flightC.capacity -
flightC.reservedSeats
and flightA.canceled = flightC.canceled
```

```
Context FlightC ::
capacity init: 300
reservedSeats init: 0
canceled init: false
reserve()
  pre: capacity-reservedSeats>0
                 and not canceled
  post:reservedSeats=reservedSeats@pre+1
 cancel()  pre: not canceled post: canceled
```

Figure 2: an instance of the state refinement pattern

An instance of the pattern's refinement condition:

Object-Z refinement conditions - F1 - for UML classes FlightA and FlightC via some retrieve relation R are automatically generated from the generic refinement condition established by the pattern [19], based on the definition of downward simulation in Object-Z described in [6]. Figure 3 shows the formula F1.

Initialization

$\forall \text{FlightC.STATE} \bullet \text{FlightC.}\mathit{INIT} \Rightarrow (\exists\ \text{FlightA.STATE} \bullet \text{FlightA.}\mathit{INIT} \wedge \text{R})$

Applicability (of operation reserve)

$\forall \text{FlightA.STATE} \bullet \forall \text{FlightC.STATE} \bullet \text{R} \Rightarrow (\text{pre FlightA.reserve} \Rightarrow \text{pre FlightC.reserve})$

Correctness (of operation reserve)

$\forall \text{FlightA.STATE} \bullet \forall \text{FlightC.STATE} \bullet \forall \text{FlightC.STATE'} \bullet$

$\text{R} \wedge \text{pre FlightA.reserve} \wedge \text{FlightC.reserve} \Rightarrow \exists \text{FlightA.STATE'} \bullet \text{R'} \wedge \text{FlightA.reserve}$

Figure 3: an instance of the refinement condition for the state refinement pattern

The transformation process from Object-Z to OCL:

Then, Object-Z refinement condition - F1 - is automatically transformed into OCL expression – F1' - by applying the transformation ***T*** in the context of a UML model M1. Apart from producing an OclExpression, ***T*** returns an OclFile containing additional definitions, which are created during the transformation process (see the appendix). The main features of the transformation are as follows,

***Highlight** #1: the Object-Z retrieve relation R is replaced by its OCL counterpart.*

Graphically, the abstraction mapping (i.e., the retrieve relation) 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. On the Z side, the context of the abstraction mapping is the combination of the abstract and the concrete states (i.e., A.STATE $\wedge$ C.STATE); however, a combination of Classifiers is not an OCL legal context. Our solution consists in translating the mapping into an OCL formula in the context of the abstract classifier, in the following way:

```
Context flightA:FlightA def :
mapping(flightC : FlightC):Boolean =
flightA.freeSeats= flightC.capacity -
flightC.reservedSeats and
flightA.canceled= flightC.canceled
```

As a convention, class names in lower case are used to denote instances. It is worth mentioning that the mapping definition could alternatively have been translated into a formula in the context of the concrete classifier.

***Highlight** #2: Object-Z expression INIT is expressed in terms of an OCL boolean operation* `isInit()`*.*

A query operation `isInit()`is automatically built from the specification of the attribute's initial values included in the UML class diagram. It returns *true* if all of the instance's attributes satisfy the initialization conditions. For example:

```
context FlightA def: isInit(): Boolean =
self.freeSeats = 300 and self.canceled = false

context FlightC def: isInit(): Boolean =
self.capacity=300 and self.canceled=false and
self.reservedSeats=0
```

***Highlight** #3: expressions containing the Object-Z operator "pre" are translated into the corresponding OCL pre conditions from the UML model.*

For example, the Object-Z expression "**pre** `FlightA.reserve"` is translated into `"flightA.freeSeats>0` **and not** `flightA.canceled"`

While, the expression "**pre** `FlightC.reserve"` is translated into `"flightC.capacity-flightC.reservedSeats>0` **and not** `flightC.canceled"`

***Highlight** #4: Object-Z expressions containing operation's invocations are translated to OCL post conditions from the UML model.*

In Object-Z, elements belonging to the pre-state are denoted by undecorated identifiers, while elements in the post-state are denoted by identifiers with a decoration (i.e. a stroke). In OCL the naming convention goes exactly in the opposite direction, that is to say, undecorated names refer to elements in the post-state. Then, in order to be consistent with the rest of the specification, a decoration (i.e., "_post") is added to each undecorated identifier in the post condition and the original decoration (i.e., @pre) is removed from the rest of the identifiers. For example the following definition:

```
Context flightA:FlightA::reserve()
 post: flightA.freeSeats= flightA.freeSeats@pre -1
```

is renamed in the following way:

```
Context flightA:FlightA::reserve()
 post: flightA_post.freeSeats= flightA.freeSeats -1
```

***Highlight #5:** logic connectors and quantifiers are translated to OCL operators.*

The Z expression ∀S.*STATE*•exp is translated to `S.allInstances() -> forAll (s | T(expr))`. The Z expression ∃S.STATE•exp is translated to S.allInstances()-> `exists(s| T(expr))`.

Notice that the name of the class, in lower case, is used to name the iterate variable. Finally, the symbol ⇒ is translated to ***implies*** and the symbol ∧ is translated to ***and.***

The appendix contains the formal definition of transformations **T** from Object-Z refinement conditions to OCL expressions. On top of that formalization the transformation process was fully automated. Table 1 shows the formula F1' that is the result of applying the transformation **T** on both the UML model M1 (figure 2) and the Object-Z refinement conditions F1 (figure 3).

Table 1: OCL refinement conditions for an instance of the state refinement pattern.

	OCL refinement condition

Initialization	FlightC.allInstances()->***forAll***(flightC| flightC.isInit()
	implies (FlightA.allInstances()-> ***exists***(flightA|
	flightA.isInit()*and* flightA.mapping(flightC)))))

Applicability	FlightA.allInstances-> ***forAll***(flightA| FlightC.allInstances-> ***forAll***(flightC| flightA.mapping(flightC) ***implies*** (flightA.freeSeats>0 *and* *not* flightA.canceled ***implies*** flightC.capacity- flightC.reservedSeats>0 ***and not*** flightC.canceled)))
Correctness	FlightA.allInstances()-> ***forAll***(flightA| FlightC.allInstances() -> ***forAll***(flightC| FlightC.allInstances()-> ***forAll***(flightC_post| flightA.mapping(flightC)***and*** (flightA.freeSeats>0 *and* *not* flightA.canceled) ***and*** (flightC_post.reservedSeats = flightC.reservedSeats+1) ***implies*** FlightA.allInstances()-> ***exists***(flightA_post| flightA_post.mapping(flightC_post) ***and*** flightA_post.freeSeats= flightA.freeSeats -1))))

4. Micromodels for evaluating refinement conditions

Generally, UML models specify an infinite number of instances; even little models such as the one described in figure 2 (i.e., there is an infinite number of instances of the type FlightA and an infinite number of instances of the type FlightC); thus to decide whether a certain property holds or not in the model results generally unfeasible.

In order to make the evaluation of refinement conditions viable, the technique of micromodels (or micro-worlds) of software is applied by defining a finite bound on the size of instances and then checking whether all instances of that size satisfy the property under consideration (i.e., the refinement condition):

- If we get a positive answer, we are somewhat confident that the property holds in all instantiations. In this case, the answer is not conclusive, because there could be a larger instantiation which fails the property, but nevertheless a positive answer gives us some confidence.

- If we get a negative answer, then we have found an instantiation which violates the property. In that case, we have a conclusive answer, which is that the property does not hold in the model.

Jackson's small scope hypothesis [9] states that negative answers tend to occur in small worlds already, boosting the confidence we may have in a positive answer.

For example, we will consider micro-worlds of the UML model in figure 2 containing only three instances of Integer and one instance of Boolean. Then we will check whether all micro-worlds of that size satisfy the refinement condition, that is to say:

Applicability Condition for operation reserve():

```
Set{ <0,f>,<1,f>,<2,f> }-> forAll (flightA|
Set{<0,0,f>,<0,1,f>,<0,2,f>,<1,0,f>,<1,1,f>,<1,2,f>,<2,0,f>,
    <2,1,f>,<2,2,f>} ->forAll(flightC|
    flightA.mapping(flightC) implies
    (flightA.freeSeats>0 and not flightA.canceled
     implies flightC.capacity-flightC.reservedSeats>0 and
                 not flightC.canceled )))
```

This expression can be easily evaluated by an ordinary OCL evaluator, returning a positive answer, which gives us some confidence that the property holds.

Lets explore a case where the refinement conditions are not satisfied; lets consider for example that preconditions were strengthened in class FlightC,

```
Context flightC:FlightC :: reserve()
pre: flightC.capacity- flightC.reservedSeats>2
     and not flightC.canceled
```

Then, the property to be checked is as follows,

```
Set{ <0,f>,<1,f>,<2,f> }-> forAll (flightA|
Set{<0,0,f>,<0,1,f>,<0,2,f>,<1,0,f>,<1,1,f>,<1,2,f>,<2,0,f>,
    <2,1,f>,<2,2,f>}  ->forAll(flightC|
flightA.mapping(flightC) implies
(flightA.freeSeats>0 and not flightA.canceled
 implies flightC.capacity-flightC.reservedSeats>2 and
              not flightC.canceled )))
```

which evaluates false in any micro-world such that flightA=<2,f> and flightC=<2,0,f> because of the fact that:

```
 flightA.mapping(flightC)                        holds,
(flightA.freeSeats>0 and not flightA.canceled)   holds,
(flightC.capacity - flightC.reservedSeats > 2)   does not hold.
```

Thus, the presence of such micro-worlds gives us the conclusive answer that the refinement property does not hold in the UML model.

6. Conclusion

Abstraction is a cognitive means by which software engineers deal with complexity. The idea promoted by most software development methodologies is to use models at different levels of abstraction; a series of transformations are performed starting from an abstract platform-independent model with the aim of making the model more specific at each step. Each transformation step should be amenable to formal verification in order to guarantee the correctness of the final product.

However, verification activities require the application of formal modeling languages with a complex syntax and semantics and need to use complex formal analysis tools; therefore they are rarely used in practice.

To facilitate the verification task we developed an automatic method for creating refinement conditions for UML models, written in the friendly and well-accepted OCL language. The inclusion of verification in ordinary software engineering activities will be propitiated by avoiding the application of unfamiliar languages and tools.

To complement such method, we adapted a strategy for reducing the search scope in order to make the evaluation of refinement conditions feasible. Since the satisfiable formulas that occur in practice tend to have small models, a small scope usually suffices and the analysis is reliable.

7. References

[1] Astesiano E., Reggio G. An Algebraic Proposal for Handling UML Consistency", Workshop on Consistency Problems in UML-based Software Development. UML Conference (2003).

[2] Back, R. & 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., editors -TUM-I0323, Technische Universitat Munchen. (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 70,3, Elsevier, 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 Int. Conference on Foundation of Software Engineering. Vienna. (2001).

[8] Gogolla , Martin, Bohling, Jo¨rn and Richters, Mark. Validation of UML and OCL Models by Automatic Snapshot Generation. In G. Booch, P.Stevens, and J. Whittle, editors, Proc. 6th Int. Conf. Unified Modeling Language (UML'2003). Springer, Berlin, LNCS 2863, (2003).

[9] Jackson, Daniel, Shlyakhter, I. and Sridharan. A micromodularity Mechanism. In proceedings of the ACM Sigsoft Conference on the Foundation of Software Engineering FSE'01. (2001).

[10] Kim, S. and Carrington, D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML'99 Conference, Lecture Notes in Computer Sciencie 1723 (1999).

[11] Lano,K. The B Language and Method. FACIT. Springer, (1996).

[12] Lano,K., Biccaregui,J., Formalizing the UML in Structured Temporal Theories, 2nd. ECOOP Workshop on Precise Behavioral Semantics, TUM-I9813, Technische U. Munchen (1998).

[13] Ledang, Hung and Souquieres, Jeanine. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. Procs. of IEEE Asia-Pacific Software Engineering Conference 2002. December 4-6, (2002).

[14] Liu, Z., Jifeng H., Li, X. Chen Y. Consistency and Refinement of UML Models. 3er Workshop on Consistency Problems in UML-based Software Development III, event of the UML Conference, (2004).

[15] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification.. http://www.omg.org. August 2003

[16] OCL 2.0. OMG Final Adopted Specification. October 2003.

[17] Pons C., Giandini R., Pérez G., et al. Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. In "UML Modeling Languages and Applications: Satellite Activities". Lecture Notes in Computer Science 3297. Springer, (2004).

[18] Pons Claudia. Heuristics on the Definition of UML Refinement Patterns. 32nd International Conference on Current Trends in Theory and Practice of Computer Science. SOFSEM (SOFtware SEMinar). January 21 - 27, 2006 . Merin, Czech Republic. Published in the Springer LNCS (Lecture Notes in Computer Science) by Springer-Verlag. (2006)

[19] Pons Claudia. On the definition of UML refinement patterns. Workshop MoDeVa at ACM/IEEE 8th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS) Jamaica. October 2005.

[20] Richters, Mark and Gogolla, Martin. OCL-Syntax, Semantics and Tools. in Advances in Object Modelling with the OCL. Lecture Notes in Computer Science number 2263. Springer. (2001).

[21] Smith, Graeme. The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers. ISBN 0-7923-8684-1. (2000)

[22] Van Der Straeten, R., Mens,T., Simmonds, J. and Jonckers,V. Using description logic to maintain consistency between UML-models. In Proc. 6th International Conference on the Unified Modeling Language. Lecture Notes in Computer Science number 2863. Springer. (2003).

APPENDIX: transformation from Object-Z to OCL refinement conditions

Grammar for Z refinement expressions:

This section describes the grammar for Z refinement expressions, which is a subset of Object-Z grammar presented in [21].

The grammar description uses the EBNF syntax, where terminal symbols are displayed in bold face. Optional constructs are enclosed by slanted square brackets [].

```
Predicate::=        ∃ SchemaText • Predicate
                    | ∀ SchemaText • Predicate
                    | Predicate1
Predicate1::=       className.INIT
                    | pre operationName
                    | operationName
                    | relationName
                    | Predicate1 ∧ Predicate1
                    | Predicate1 ⇒ Predicate1
                    | (Predicate)
SchemaText::=       className.STATE [Decoration]
className::=        Word
operationName::=    className.Word
relationName::=     Word [Decoration]
Word                category for undecorated names
Decoration::=       '
```

Definition for the Transformation:

This section contains the specification of function ***T*** that takes a refinement condition written in Object-Z and returns the corresponding refinement condition written in OCL. Function ***T*** is applied in the context of a UML model **M** containing all the elements which are referred to in the Z expressions. Apart from producing an OclExpression, function ***T*** returns an OclFile containing additional definitions that are created during the transformation.

UML elements are retrieved form **M** by using standard lookup operations on its environment as it is defined in [16].

T: Model -> Predicate -> (OclExpression, OclFile)

```
$T_M$(Predicate1 ∧ Predicate2)= (e,Φ)
    Where
    $T_M$(Predicate1)= (e1, Φ1)
    $T_M$(Predicate2)= (e2,Φ2)
    e= e1 "and" e2
    Φ = Φ1 merge Φ2
$T_M$(Predicate1 ⇒ Predicate2)= (e,Φ)
    Where
    $T_M$(Predicate1)= (e1, Φ1)
    $T_M$(Predicate2)= (e2,Φ2)
    e= e1 "implies" e2
    Φ = Φ1 merge Φ2
$T_M$( ∀ className.STATE • Predicate) = (e,Φ)
    Where
    $T_M$(Predicate)= (e1, Φ)
    e=className".allInstances()->forAll
    ("iteratorName"|"e1")"
    iteratorName= toLowerCase(className)
$T_M$( ∀ className.STATE' • Predicate) = (e,Φ)
    Where
    $T_M$(Predicate)= (e1, Φ)
    e=className".allInstances()->forAll
    ("iteratorName"|"e1")"
    iteratorName= toLowerCase(className) "_post"
$T_M$( ∃ className.STATE • Predicate) = (e,Φ)
    Where
    TM(Predicate)= (e1, Φ)
    e=className".allInstances()->exists
    ("iteratorName"|"e1")"
    iteratorName= toLowerCase(className)
TM( ∃ className.STATE' • Predicate) = (e,Φ)
    Where
    TM(Predicate)= (e1, Φ)
    e=className".allInstances()-
    >exists("iteratorName"|"e1")"
    iteratorName= toLowerCase(className) "_post"
TM (className.INIT) =(e,Φ)
    Where
    e= toLowerCase(className) ".isInit()"
    Φ = "Package" packageName
       "context" className "def: isInit(): Boolean ="

    propertyName1"="exp1"and"..."and"propertyNamen"="expn
       "endPackage"
    Where
    packageName = class.package.name
    class=M.getEnvironmentWithParents().lookup(className)
    Properties = class.allProperties()->select
    (p|p.initialValue->notEmpty())
    ∀j•1≤j≤properties->size()•
      propertyNamej = properties->at(j).name
      expj = properties->at(j).initialValue.body
TM(pre className.operationName) = (e, ∅)[1]
    Where:
    e = operation.precondition.specification.body
    Where:
    operation : UMLOperation =
      M.getEnvironmentWithParents().lookup(className).
      getEnvironmentWithParents()
      .lookupImplicitOperation(operationName, Sequence{})
TM (className.operationName)= (e, ∅)
    Where:
```

[1] In this document the symbol ∅ is an abbreviation denoting the empty package.

```
e =
operation.postcondition.specification.body.renamed()
Where:
operation : UMLOperation  =
  M.getEnvironmentWithParents().lookup(className).
  getEnvironmentWithParents()
   .lookupImplicitOperation(operationName,Sequence{})
```

Where:

function renamed() is applied on an OclExpression returning a copy of the expression where any undecorated name v has been renamed as v_post and any decorated name v@pre has been renamed as v.

$\mathcal{T}_M$ (relationName) = (e,Φ)

```
Where:
relationName ∈ Word -- it is an undecorated name
e = absInstance ".mapping(″ refInstance ")″
Φ = "Package″ packageName
   "Context″ absInstance ":″ AbstractClass  "def:″
   "mapping(″refInstance":″RefinedClass ″):Boolean =″
exp    "endPackage″
Where:
packageName = d.package.name
d : Abstraction =
M.getEnvironmentWithParents().lookup(relationName)
AbstractClass  = d.supplier.name
RefinedClass = d.client.name
absInstance = toLowerCase(AbstractClass)
refInstance = toLowerCase(RefinedClass)
exp = d.mapping.body
```

$\mathcal{T}_M$ (relationName′) = (e,∅)

```
Where:
e = absInstance ".mapping(″ refInstance ")″
Where:
d : Abstraction =
M.getEnvironmentWithParents().lookup(relationName)
AbstractClass  = d.supplier.name
RefinedClass = d.client.name
absInstance = toLowerCase(AbstractClass) "_post″
refInstance = toLowerCase(RefinedClass)  "_post″
```