An OCL-Based Technique for Specifying and Verifying Refinement-Oriented Transformations in MDE

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

[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. Despite the fact that the refinement technique is one of the cornerstones of a formal approach to software engineering, the concept of refinement in model driven engineering is loosely defined and open to misinterpretations. In this article we present a rigorous technique for specifying and verifying frequently occurring forms of refinement that take place in software modeling. Such strategy uses the formal language Object-Z as a background foundation, whereas designers only have to deal with the broadly accepted UML and OCL languages, thus propitiating the inclusion of verification in ordinary software engineering activities, increasing in this way the level of confidence on the correctness of the final product. Finally, an automatic tool is provided to support such model refinement activities; this tool adopts the micromodels strategy to reduce the search scope, making the verification process feasible.

1 Introduction

The idea promoted by *model-driven engineering* (MDE) [7] [24] [17] 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. Such transformations reduce non-determinism by making design decisions, e.g., how to represent data, how to implement communications, etc. In MDE predefined transformations, written in a standard transformation language [21] are applied in order to evolve from model to model. It is assumed that such transformations have been previously validated by a MDE expert, and thus are safe to apply; such transformations are *refinements* in the sense of formal languages: refinement is the process of developing a more detailed design or implementation from an abstract specification through a sequence of mathematically-based steps that maintain correctness with respect to the original specification.

Despite the fact that refinement technique is one of the cornerstones of a formal approach to software engineering, the concept of refinement in MDE is loosely defined and open to misinterpretations. This drawback takes place because of the semi-formality present in the modeling languages used in MDE and also because of the currently relative immaturity in this field.

O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 646 – 660, 2006.

There are two alternatives to increase the robustness of the MDE refinement machinery. One is to translate the core language used in MDE, i.e., UML [15], into a formal language such as Z, where properties are defined and analyzed. For example the works presented in [1], [3], [5], [10], [11], [13] and [25] 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) 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 and undecidedly. A second alternative is to promote a formal definition of refinement, e.g., simulation in Z, and express it in MDE terms. For example, Boiten and Bujorianu in [2] indirectly explore refinement through unification; Paige and colleges in [18] define refinement in terms of model consistency; Liu, Jifeng, Li and Chen in [14] 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. Finaly, Lano and colleges in [12] describe a catalogue of *UML refinement patterns* which is a set of rules to systematically transform UML models to forms closer to Java code.

Following the second alternative, in [19] and [20] 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. In this article we work further on such proposal by enriching those refinement patterns with a refinement condition written in OCL (Object Constraint Language) [16] [22]. The advantage of this approach is that refinement conditions get completely defined in terms of OCL, making the application of languages which are usually hardly accepted by software engineers unnecessary. 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 related work, conclusions and future projects.

2 Refinements Specification and Verification in Object-Z and UML

In Object-Z [23], 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: an 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.

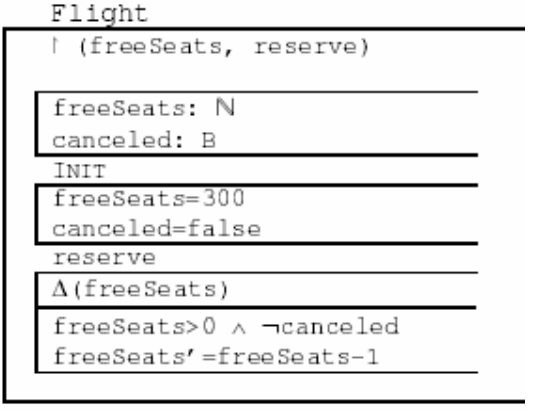

Fig. 1. Simple Object-Z schema

Object-Z is equipped with a schema calculus (i.e., a set of operators provided to manipulate Object-Z schemas). The schema calculus makes it possible to create Object-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 = {⟨freeSeats=x, canceled=t⟩ | 0≤x≤300 ∧ t∈{true, false}}

- Operator INIT denotes the initial states of a given schema. For example, Flight.INIT={⟨freeSeats=300 , canceled=false⟩ | }

- 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 = {⟨freeSeats=x, canceled=false ⟩ | 0<x≤300}

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

- Schema implication (S ⇒ T) denotes the usual logical implication.

In [4], 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) ∀C.STATE • C.INIT ⇒(∃A.STATE • A.INIT ∧ R)

(Applicability) ∀A.STATE • ∀C.STATE• R ⇒ (pre A.op ⇒ pre C.op)

(Correctness) ∀A.STATE•∀C.STATE•∀C.STATE'•

R ∧ pre A.op ∧ C.op ⇒ ∃A.STATE'• R' ∧ 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 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 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 [12] [19] [20] document recurring refinement structures in UML models. In this section we present a process to be applied on UML models containing such patterns in order to automatically create OCL refinement conditions to analyze them in a rigorous way. Figure 2 gives a description of the process at a glance. It is based on a pipeline architecture in which the analysis is carried out by a sequence of steps. The output of each step provides the input to the next one. In this section we give a brief overview of each step:

Refinement pattern instantiation. Each refinement pattern P consists of two parts: a description M of the Pattern structure, given in terms of UML diagrams and a generic constraint F expressed in Object-Z representing refinement condition for such pattern. Given a UML model M1 compliant with the structure of pattern P, the first step of the process automatically generates an instance F1 of the generic formula F that establishes the conditions to be fulfilled by M1 in order to verify the refinement.

Transformation to OCL. After being generated, the Object-Z formula F1 is automatically translated into the OCL formula F1' by applying the transformation **T** (the detailed definition of **T** is included in the appendix).

Micromodels strategy application. In this step, the micromodels strategy is applied to F1' in order to produce a formula F1'' which is analyzable within a limited scope.

OCL Evaluation. Finally, F1'' is submitted to an ordinary OCL evaluator.

The process is assisted by the automated tool ePlatero [6] that is a plug-in to the Eclipse development environment; ePlatero implements the verification process described above.

In the following sections this process is illustrated through a concrete example: the state refinement pattern [19].

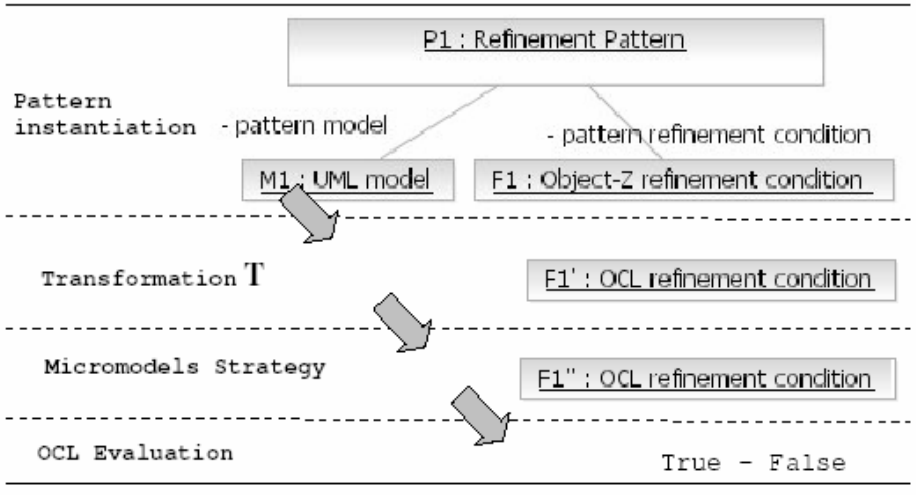

Fig. 2. Overview of the refinement verification process

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 3, 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] has been used to specify initial values, operation's pre and post conditions and the mapping attached to the refinement relationship.

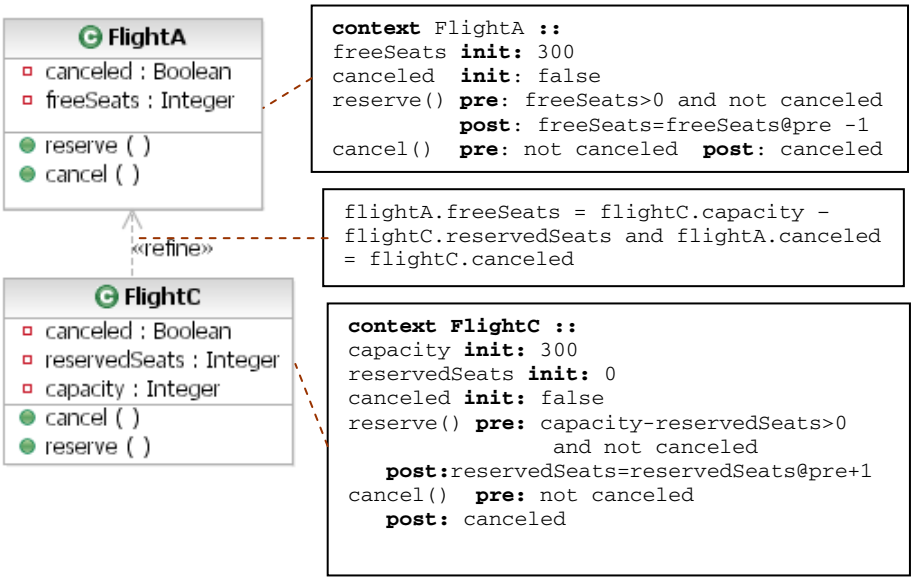

Fig. 3. an instance of the state refinement pattern

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 [4]. Figure 4 shows the formula F1.

Initialization

$\forall$FlightC.STATE $\bullet$FlightC.*INIT* $\Rightarrow$($\exists$ FlightA.STATE $\bullet$FlightA.*INIT* $\wedge$ R)

Applicability (for operation reserve)

$\forall$FlightA.STATE$\bullet\forall$FlightC.STATE $\bullet$

R $\Rightarrow$ (pre FlightA.reserve $\Rightarrow$ pre FlightC.reserve)

Correctness (for operation reserve)

$\forall$FlightA.STATE $\bullet\forall$FlightC.STATE $\bullet$ $\forall$FlightC.STATE'$\bullet$ R $\wedge$

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

Fig. 4. 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 $\boldsymbol{T}$ 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:

```
T : Model-> ObjectZpredicate -> (OclExpression,OclFile)
```

The main features of the transformation are as follows,

Remark #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 ∧ 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.

Formally*:*

T_M (relationName) = (e,Φ)

Where:

e = absInstance "**.mapping(**" refInstance ")"

Φ = "**package**" packageName
"**context** a**:**" AbstractClass "**def**:"
"mapping(c:" RefinedClass "):Boolean =" exp
"endPackage"

Where:

d = M.getEnvironmentWithParents().lookup(relationName)
AbstractClass = d.supplier.name
RefinedClass = d.client.name
absInstance = toLowerCase(AbstractClass)
refInstance = toLowerCase(RefinedClass)
exp = d.mapping.body
packageName = abstractClass.package.name

Remark #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
```

In cases where the refinement involves composite classes, the initialization condition is built in terms of the initialization of each component; additionally, information provided for each composite association (e.g., multiplicity) is taken into consideration.

Formally:

τ_M (className.*INIT*) =(e,Φ)

Where

e= toLowerCase(className) ".isInit()"

Φ = "**Package**" $packageName_1$

"**context**" className "**def**: isInit(): Boolean ="

$attName_1$"="exp_1"**and**"..."**and**" $attName_n$"="exp_n "**and**"
$navigationName_1$"->size() =" $size_1$ "**and**"
$navigationName_1$ "->forAll(p| p.isInit())"..."**and**"
$navigationName_n$"->size() =" $size_n$ "**and**"
$navigationName_n$ "->forAll(p| p.isInit())"
"**endPackage**"

Where

packageName = class.package.name

class : UMLClass =
M.getEnvironmentWithParents().lookup(className)

attributes: Sequence(UMLProperty) =
class.allProperties()->select(p|p.initialValue->notEmpty())

$\forall j \times 1 \leq j \leq$attributes->size()• $attName_j$ = attributes->at(j).name ∧ exp_j = attributes->at(j).initialValue.body

navigations: Sequence(UMLProperty) =
class.allProperties()->select(p|p.association->notEmpty() and p.isComposite())

$\forall j \times 1 \leq j \leq$navigations->size()• $navigationName_j$ = navigations->at(j).name ∧ $size_j$ = navigations->at(j).lower

Remark #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`"

Remark #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::reserve()
 post: self.freeSeats= self.freeSeats@pre -1
```

is renamed to:

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

Remark #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))`. While the Z expression ∃`S.STATE•exp` is translated to `S.allInstances()->exists(s| T(expr))`.

For example, the translation for the universal quantifiers is as follows:

```
T_M( ∀ className.STATE • Predicate) = (e,Φ)
    Where
    T_M(Predicate)= (e1, Φ)
    e=className".allInstances()->forAll("iteratorName"|"e1")"
    iteratorName= toLowerCase(className)
```

Notice that the name of the class, in lower case, is used to name the iterate variable. Finally, the symbol $\Rightarrow$ is translated to ***implies*** and the symbol $\wedge$ is translated to ***and***,

$\mathbf{T_M}$(Predicate1 $\wedge$ Predicate2)= (e,Φ)

Where

$\mathbf{T_M}$(Predicate1)= (e1, Φ1)

$\mathbf{T_M}$(Predicate2)= (e2, Φ2)

e= e1 "**and**" e2

Φ = Φ1 *merge* Φ2

On top of the formal definition of **T** the transformation process was fully automated [6]. Table 1 shows the formula F1' that is the result of applying the transformation **T** on both the UML model M1 (figure 3) and the Object-Z refinement conditions F1 (figure 4).

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()-> f*orAll*(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))))

3.2 Further Patterns

A vast number of refinement patterns can be specified and verified following the method described in the preceding section, for example:

- *Object decomposition refinement pattern* is a form of refinement in which an abstract element is described in more detail by revealing its interacting internal components and conversely, the composite represents its components in sufficient detail in all contexts in which the fact of being composed is not relevant.
- *Atomic operation refinement pattern* occurs in the case that a more concrete specification is obtained from an abstract specification by replacing any operation Aop_k by its refinement Cop_k. The refined operation reduces non-determinism and/or partiality present in the abstract operation.
- *Non-atomic operation refinement pattern* 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. Non-atomic refinements are useful because they allow the initial specification to be described independently of the structure of the eventual implementation. Also it enables considerations of efficiency to be gradually introduced.
- *Promotion pattern* illustrates an elegant relationship between promotion and refinement, under certain circumstances the promotion of a refinement is a refinement of a promotion [4].

Additionally, complex model transformations, such as the application of most GoF design patterns and the use of refactoring can be specified as a composition of the simpler patterns described above.

4 Micro-worlds for Evaluating Refinement Conditions

Even little models such as the one described in figure 3 specify an infinite number of instances; 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:

- If we get a positive answer, we are somewhat confident that the property holds in all worlds. In this case, the answer is not conclusive, because there could be a larger world which fails the property, but nevertheless a positive answer gives us some confidence.
- If we get a negative answer, then we have found a world 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 already tend to occur in small worlds, boosting the confidence we may have in a positive answer. For example, in order to generate suitable micro-worlds to evaluate the refinement conditions of class diagram in figure 3, the OCL package shown in figure 5, containing invariants that reduce the size of the micro-world, is provided.

```
package flights
    context FlightA
        inv: Set { 0 .. 300 } -> includes (self.freeSeats)
    context FlightC
        inv: Set {300} -> includes (self.capacity)
        inv: self.reservedSeats <= self.capacity
 endpackage
```

Fig. 5. OCL invariants reducing the search space

Apart from satisfying all the OCL invariants reducing the search space, to be suitable to analyze refinement relationships, the micro-worlds should satisfy the "*duality property*". Such property establishes that for each instance of a concrete class there must exist at least an instance of the abstract class being related by the abstraction mapping. The automatic micro-world generation process implemented by the tool guarantees the fulfillment of the duality property.

Then the tool checks whether all micro-worlds of that size satisfy the refinement condition. For example, figure 6 displays one of the micro-worlds satisfying the invariants and the duality property. In such micro-world the expression FlightA.allInstances() returns a finite set of size three containing the objects FlightA1, FlightA2 and FlightA3, while FlightC.allInstances() returns a finite set of size three containing the objects FlightC1, FlightC2 and FlightC3.

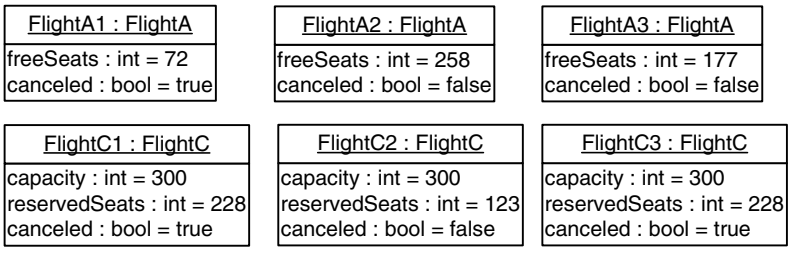

Fig. 6. Micro-world automatically generated from the UML model in figure 3 enriched with the constraints in figure 5

In this context we have, for example, the following applicability condition for operation `reserve()`:

```
 Set{<FlightA1>, <FlightA2>, <FlightA3>} -> forAll (flightA |
 Set{<FlightC1>, <FlightC2>, <FlightC3>}  -> 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 is 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 as follows,

```
context FlightC :: reserve()
pre:self.capacity-self.reservedSeats>200 and notself.canceled
```

Then, the property to be checked would be,

```
 Set{<FlightA1>, <FlightA2>, <FlightA3>} -> forAll (flightA |
 Set{<FlightC1>, <FlightC2>, <FlightC3>} -> forAll(flightC |
flightA.mapping(flightC) implies (flightA.freeSeats>0 and not
flightA.canceled implies flightC.capacity -
flightC.reservedSeats >200 and not flightC.canceled)))
```

which evaluates false in the micro-world in figure 6, as follows:

```
flightA3.mapping(flightC2)= true
flightA3.freeSeats>0 and not flightA3.canceled = true
flightC2.capacity – flightC2.reservedSeats > 200  = false
```

giving the conclusive answer that the refinement property does not hold in this last model.

5 Conclusion

Each transformation step in the model driven software development process 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 standard and well-accepted OCL language. This is a lightweight approach that avoids the use of mathematical languages and tools that while ideal and suitable for the problem, will likely be unacceptable to developers.

The inclusion of verification in ordinary software engineering activities will be propitiated by encouraging the use of tools that are familiar and usable to MDE developers. The disadvantages of this approach relate to soundness and completeness; while the approach is rigorous, it is not formal and thus it is not possible to verify that the definition is sound and complete.

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.

Acknowledgement. This work was partially funded by Universidad Abierta Interamericana (UAI), through the project ?Software modelling: a formal approach?.

References

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

[2] 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).

[3] Davies J. and Crichton C. Concurrency and Refinement in the Unified Modeling Language. Electronic Notes in Theoretical Computer Science 70,3, Elsevier, 2002.

[4] Derrick, J. and Boiten,E. Refinement in Z and Object-Z. Foundation and Advanced Applications. FACIT, Springer. (2001)

[5] 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. (2001).

[6] ePlatero. http://sol.info.unlp.edu.ar/eclipse.

[7] Favre Jean-Marie, Estublier Jacky, Blay Mireille. Beyond MDA : Model Driven Engineering (L'Ingénierie Dirigée par les Modèles : au-délà du MDA) Edition Hezmes-Lavoisier, ISBN 2-7462-1213-7. Février 2006.

[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, 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., Biccaregui,J., Formalizing the UML in Structured Temporal Theories, 2nd. ECOOP Workshop on Precise Behavioral Semantics, TUM-I9813, (1998).

[12] Lano, Kevin, Androutsopolous, Kelly and Clark David. Refinement Patterns for UML. Proceedings of REFINE'2005. Elsevier Electronic Notes in Theoretical Computer Science 137. pages 131-149 (2005).

[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] OMG - 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] Object Management Group, *MDA Guide*, v1.0.1, omg/03-06-01, June 2003.

[18] Paige, R., Kolovos D. and Polack,F. Refinement via Consistency Checking in MDD. In REFINE'2005. Electronic Notes in Theoretical Computer Science 137. (2005).

[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] Pons Claudia. Heuristics on the Definition of UML Refinement Patterns. 32nd International Conference on Current Trends in Theory and Practice of Computer Science. SOFSEM. Lecture Notes in Computer Science LNCS number 3831. Springer (2006)
[21] QVT Partners revised submission to QVT 1.1 (ad/2003-08-08).
[22] 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).
[23] Smith, Graeme. The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers. ISBN 0-7923-8684-1. (2000)
[24] Stahl, M Voelter. Model Driven Software Development. John Wiley, ISBN 0470025700, April 2006.
[25] 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).