Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 220 (2008) 43–61
www.elsevier.com/locate/entcs

A Lightweight Approach for the Semantic Validation of Model Refinements

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

LIFIA. Facultad de Informática
Universidad Nacional de La Plata
Buenos Aires, Argentina

Abstract

Model Driven Engineering proposes the use of models at different levels of abstraction. Step by step validation of model refinements is necessary to guarantee the correctness of the final product with respect to its initial models. But, given that accurate validation activities require the application of formal modeling languages with a complex syntax and semantics and need to use complex formal analysis tools, they are rarely used in practice. In this article we describe a lightweight validation approach that does not require the use of third-party (formal) languages. The approach makes use of the standard OCL as the only visible formalism, so that refinements can be checked by using tools that are fully understood by the MDE community. Additionally, for the efficient evaluation of the refinement conditions a hybrid strategy that combines model checking, testing and theorem proving is implemented. Correctness and complexity of the proposal are empirically validated by means of the development of case studies and a comparison with the Alloy analyzer.

Keywords: modeling, refinement, model transformation, Object Constraint Language, OCL, MOF, UML, validation, testing, model checking.

1 Introduction

The idea promoted by model-driven software engineering (MDE) [17] [34] [26] 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 step. Currently, a considerable amount of research is being performed in the area of model transformations, as evidenced by the development of a number of different model transformation approaches, including ATL [15] and TefKat [20], based on the QVT standardization initiative [30]. That research mainly focuses on how to express model transformations (defining a model

[1] This research was funded in part by the UAI (Universidad Abierta Interamericana) and in part by the CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)

[2] Email: cpons@sol.info.unlp.edu.ar

[3] Email: dgarcia@sol.info.unlp.edu.ar

doi:10.1016/j.entcs.2008.11.005

transformation language), while less attention was paid on how to systematically validate model transformations (defining validation criteria for model transformation). In general, the validation of model transformation may include properties such as syntactic correctness of the model transformation with regards to its specification language and syntactic correctness of the models produced by the model transformation (see for example [21] and [18]). But, few proposals deal with the semantic consistency of the model transformation, that is to say, the preservation of target model correctness with regards to the corresponding source model.

However, the problem of semantic validation of model transformation is not a new challenge originated by the MDE philosophy. The idea of software development being conducted in a controlled and provably correct fashion, in incremental steps, goes back to the days of Dijkstras stepwise refinement theory [8]. Since these seminal ideas, refinement theory has found widespread adoption and development in the formal method community, where the majority of refinement schemes revolve around the principle of "substitutivity" [7]. In such a scheme, a refinement is deemed correct if the observable behavior of a program/model is undetectable after a refinement has occurred. Refinement is usually checked by proving that the concrete system simulates the abstract one. The notion of simulation is captured by downward and upward simulation rules comprising conditions relating the possible initializations and transitions of the concrete and abstract systems.

To adapt those well-founded refinement definitions towards the validation of model transformation becomes a tempting challenge. However, formal techniques have not been successfully applied to large-scale design models in a language such as UML [37]. There are two main reasons for this. Firstly, the general techniques do not scale well to the size required, even when techniques are used to reduce the state space. Secondly, it is beyond the expertise of most developers to write the mathematically formal statements of correctness for refinements.

In this paper we describe a novel light-weight formal approach towards the semantic validation of model refinements. The proposed approach provides a more practical approach to refinement than the strict notions found in the formal methods community. Specifically, we consider how refinements between state-based specifications (e.g., written in MOF [23] or UML class diagrams) can be checked by using tools that are fully understood by the MDE community. In particular, we show how the Object Constraint Language OCL [25] can be used to encode the standard simulation conditions. OCL is part of the standards UML and MOF and will probably form part of most modeling tools in the near future, thus the main advantage of our approach is that it does not require the use of third-party (formal) languages.

Additionally, in order to make the evaluation of refinement conditions more efficient, we implement a hybrid strategy that combines model checking, testing and theorem proving, based on the micro-worlds generation strategy presented in [14].

The structure of this document is as follows: section 2 serves as a brief introduction to the issue of refinement specification in formal languages (in particular we use the Z language as foundation) as well as in MDE languages (in particular MOF

2.0 and UML 2.0); section 3 describes the automatic method for creating OCL refinement condition for UML/MOF model refinements; section 4 describes a hybrid evaluation strategy for the efficient evaluation of refinement conditions; section 5 presents experimental results and finally the paper closes with a presentation of related work and conclusions.

2 Refinement specification and verification

In this section we briefly introduce the concept of refinement specification and verification both in terms of the formal language Z [33] and in terms of MOF and UML.

2.1 Refinement in Z

Data refinement is a formal notion of development, based around the idea that a concrete specification can be substituted for an abstract one as long as its behavior is consistent with that defined in the abstract specification. In a state-based setting, as typified by Z, data refinements are usually verified by defining a relation (referred to as a retrieve relation R) between the two specifications and verifying a set of simulation conditions. In general there are two forms the simulation rules take depending on the interpretation given to an operation (specifically, depending on the interpretation given to the operations guard or precondition). The two interpretations are often called the blocking and non-blocking semantics. We consider only the non-blocking semantics in this paper. Under this semantics, an operation has a precondition outside of which its behavior is undefined, and it is the standard semantics for sequential specification (and as such is the normal semantics for refinement in Z).

Let a specified system comprise a set of states S, a non-empty set of initial states $I \subseteq S$, and a finite set of operations $\{Op_1, ..., Op_n\}$, each of which is a relation between states in S (input and output parameters of operations can be embedded in the states of S as described by Smith and Winter in [32]). Under the non-blocking semantics, downward simulation is then defined as follows [7].

Definition 2.1 (Downward simulation: non-blocking) A specification C = (CS, CI, $\{COp_1, ..., COp_n\}$) is a downward simulation of a specification A = (AS, AI, $\{AOp_1, ..., AOp_n\}$), if there exists a retrieve relation R between AS and CS such that the following hold for all i $\in 1, ..., n$.

1 $\forall c \in CS \bullet c \in CI \Longrightarrow \exists a \in AS \bullet a \in AI \wedge aRc$

2 $\forall a \in AS; c \in CS \bullet aRc \Longrightarrow$ (pre $AOp_i \Longrightarrow$ pre COp_i)

3 $\forall a \in AS; c, c' \in CS \bullet$ (pre AOp_i) $\wedge$ aRc $\wedge$ c COp_i c $\Longrightarrow$ ($\exists a' \in AS \bullet$ a'Rc' $\wedge$ a AOp_i a')

Condition 1 is known as initialization. It requires that for every concrete initial state there is an initial abstract state related by the retrieve relation R.

Condition 2 is the applicability condition. It allows preconditions to weaken under a refinementthe concrete operation must be applicable everywhere the abstract is applicable, but can also be defined on additional states.

Condition 3 is known as correctness. It requires consistency of behavior between abstract and concrete operations, but only on those states where the abstract operation is enabled. By the applicability condition, the concrete operation may be enabled on other states, upon which no constraints are imposed. 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.

To verify a data refinement it is sometimes necessary to use an alternative simulation rule known as an upward simulation; in general, both are needed to form a complete methodology for verifying refinements (see [7]), however it is left as future work. Besides, for practical reasons we restrict our attention to systems where the inverse of R is a total function (usually referred to as abstraction function). In such cases the refinement conditions can be simplified as follows:

Let a= R^{-1} (c) and a'=R^{-1} (c') in

1 $\forall c \in CS \bullet c \in CI \implies a \in AI$

2 $\forall c \in CS \bullet$ pre $AOp_i \implies$ pre COp_i

3 $\forall c, c' \in CS \bullet$ pre $AOp_i \wedge$ c COp_i c' $\implies$ a AOp_i a'

To illustrate the topic of refinements, figure 1 (left side) displays the specification of a simple data type called FlightA, containing information about a flight booking system where each flight is abstractly described by the total capacity of the flight together with the quantity of reserved seats; a Boolean attribute is used to represent the state of the flight (open or canceled). The specification describes the initialization condition (named Init) and the two available operations: *reserve* to make a reservation of one seat and *cancel* to cancel the entire flight. Then, figure 1 (right side) shows a refinement for FlightA, named FlightC, that is obtained 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 with their own attributes and behavior (a seat has an identification number and a Boolean attribute indicating whether it is reserved or not). The refined version of the reservation process selects a seat (ready to be reserved) in a non-deterministic way.

By evaluating the three refinement conditions we are able to formally verify whether FlightC is a refinement of FlightA or not. Graeme Smith and John Derrick in [31] consider how refinements between specifications written in Z can be effectively checked by use of a model checker.

2.2 Refinement in MOF/UML

The modeling languages UML and MOF provide visual artifacts to specify the structure and behavior of object-oriented systems. UML and MOF specifications

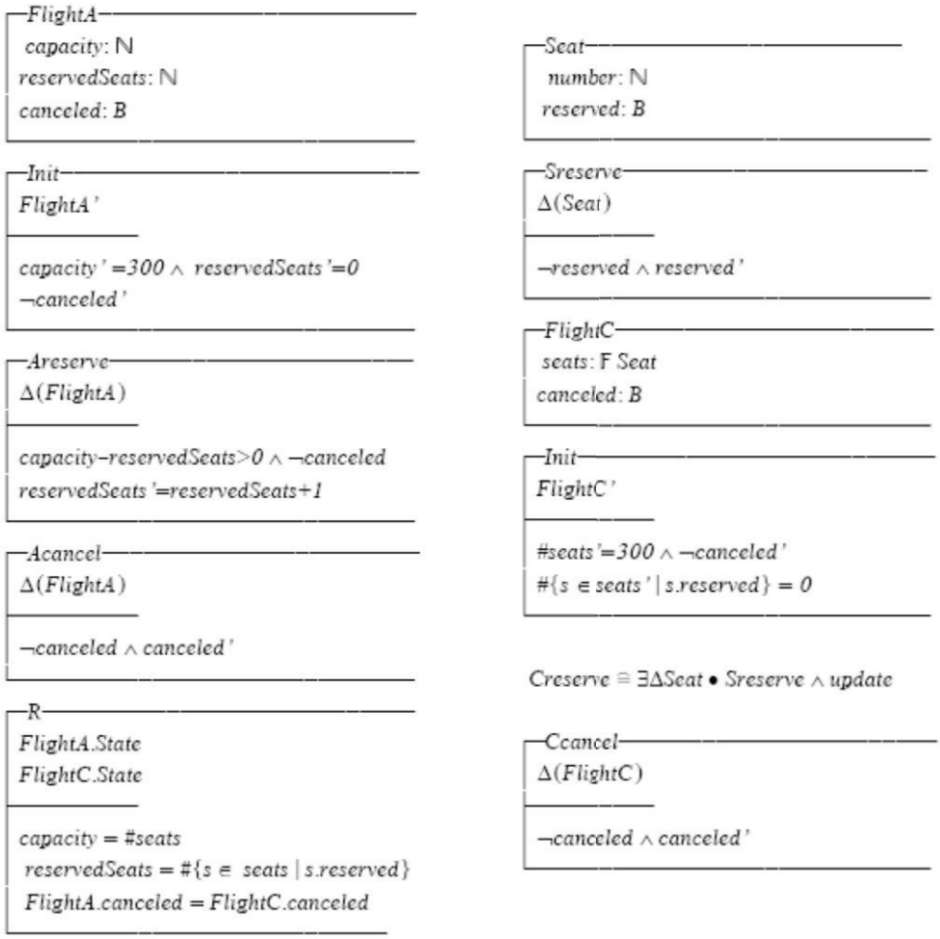

Fig. 1. Z refinement structure

share a common core infrastructure [36]. The OCL 2.0 is aligned with UML 2.0 and MOF 2.0 and it contains a well-defined and named subset of OCL that is defined purely based on the common core of UML and MOF. This allows this subset of OCL to be used with both the MOF and the UML. This common core defines a modeling artifact named DirectedRelationship to connect two (or more) related elements. Afterwards, languages based on this common core might specialize the DirectedRelationship metaclass in order to provide specific notations for specific kind of relationships. In particular, UML defines an artifact named Abstraction (a specialization of DirectedRelationship) with the stereotype `<<refine>>` to explicitly specify the refinement relationship between named model elements. The Abstraction artifact has a meta-attribute called mapping designated to record the abstraction/implementation mappings (i.e., the counterpart to the 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. These visual artifacts are illustrated in the figure 2, where the previously introduced data type named FlightA and its refinement named FlightC are specified by a UML class diagram; OCL was used to specify the pre and post conditions of the operations (see figure 3).

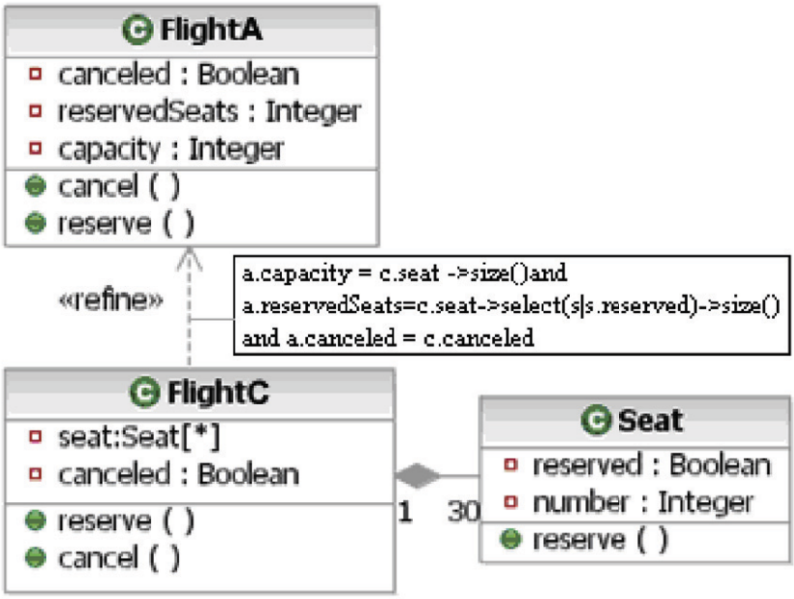

Fig. 2. Refinement specification in MOF/UML

On the semantic side, the definition of refinement in the UML specification [37] is formulated using natural language and it remains open to numerous interpretations. Therefore, MOF and UML languages are expressive enough to visually specify model refinements, but they lack formal semantics. Without a formal semantics, to carry out any verification process becomes unworkable.

To overcome this drawback we need to define a well-founded refinement theory for MOF and UML. In addition, it would be desirable for such theory to be expressible in a language compliant with MOF, in the same way as Z refinement conditions are defined in the Z itself, so that refinement evaluation could be carried out into the same development environment by using tools that are familiar to model-driven developers.

3 Validation strategy for MOF/UML refinement

In this section we discuss a general approach to checking refinement conditions for UML class diagrams. The process relies on properties of the common core infrastructure only. The process takes a class diagram as input and produces a refinement condition (written in OCL) for such UML model, as output. Specifically,

Abstract	**context** FlightA :: capacity **init:** 300 reservedSeats **init:** 0 canceled **init:** false **context** FlightA :: reserve() **pre:** not self.canceled and self.capacity - self.reservedSeats > 0 **post:** self.reservedSeats = self.reservedSeats@pre + 1 **context** FlightA :: cancel() **pre:** not self.canceled **post:** self.canceled
Concrete	**context** FlightC :: canceled **init:** false **context** Seat :: reserved **init:** false **context** FlightC :: reserve() **pre:** self.seat-> exists(q| not q.reserved) and not self.canceled **post:** self.seat-> exists(s| s.reserved and self.seat@pre-> exists(q| q.number=s.number and not q.reserved)) **context** FlightC :: cancel() **pre:** not self.canceled **post:** self.canceled **context** Seat :: reserve() **pre:** not self.reserved **post:** self.reserved

Fig. 3. Constraints for the Flights model

it encodes downwards simulation under the non-blocking semantics (as defined for Z) in OCL. Then, the refinement conditions are evaluated by the use of an OCL evaluator. The process is fully automatized by a software tool.

3.1 The retrieve relation

We begin by considering the retrieve relation. Graphically, the mapping describing the relation between the attributes in the abstract state and the attributes in the concrete state is attached to the refinement relationship. On the Z side, the context of the abstraction mapping is the combination of the abstract and the concrete states (i.e., AS and CS). Since a combination of Classifiers is not an OCL legal context, our solution consists in translating the mapping into an OCL definition in the context of the refined classifier. For example, the following function definition is automatically derived from the mapping in figure 2,

context c: FlightC **def**: abs(): FlightA = FlightA.allInstances() -> select (a | a.capacity = c.seat -> size() **and** a.reservedSeats = c.seat -> select (s| s.reserved) -> size() **and** a.canceled = c.canceled) -> any()

Given an instance of the refined classifier the function *abs()* returns its (unique) abstract representation.

3.2 Initialization condition

This condition requires that for each concrete initial state we are able to find an abstract initial state related by the abstraction function. To check whether an object is in its initial state we introduce a query operation *isInit()* which is automatically built from the specification of the attribute's initial values included in the class diagram. Composite associations are also considered during the construction of the initialization conditions. That is to say, the initialization condition is built in terms of the initialization of each component. The approach consists in collecting the attribute's initial values included in the diagram first; and then collecting the information provided for each composite association (properties of the association such as *multiplicity*, *isOrdered* and *isUnique* are taken into consideration); finally, the *isInit()* operation is invoked on each one of the components. It returns *true* if all of the attributes and components satisfy the initialization conditions. For example, the following initialization queries are automatically derived from the class diagram in figure 2.

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

context Seat **def**: isInit(): self.reserve=false

context FlightC **def:** isInit(): Boolean = self.seat -> size()=300 **and** self.canceled=false **and** self.seat -> forAll(s | s.isInit())

Thus, the following initialization condition for Z specifications,

$$\forall c \in FlightC \bullet c \in FlightCI \implies a \in FlightAI$$

is expressed in OCL by means of the following constraint, where *a=c.abs()*,

FlightC.allInstances() -> forAll(c | c.isInit() implies a.isInit())

Notice that the universal quantification "$\forall c \in FlightC$" is trivially represented by the OCL expression "FlightC.allInstances()->forAll(c |". The Z connector "$\implies$" was converted to the OCL connector "**implies**".

3.3 Applicability condition

We now consider the applicability condition. To check applicability, we need to be able to determine whether each of the abstract and concrete operations can occur. For each operation $Op_i()$ involved in the refinement a Boolean operation preOpi() is created. This operation will evaluate *true* if the precondition of the operation $Op_i()$ is fulfilled. The body of each operation $preOp_i()$ is automatically derived

from the OCL preconditions attached to the operation $Op_i()$ in the class diagram. For example, given the specifications of operation reserve in figure 2 the following (polymorphic) operations are automatically generated,

context FlightA **def**: preReserve(): Boolean = **not** self.canceled **and** self.capacity - self.reservedSeats > 0

context FlightC **def**: preReserve(): Boolean = self.seat-> exists(q | not q.reserved) **and not** self.canceled

Then, Z expressions containing the operator "pre" are represented in OCL by means of invocations to these Boolean operations. Thus, the following applicability condition

$\forall c \in CS \bullet$ pre Areserve $\Longrightarrow$ pre Creserve

is encoded in OCL by means of the following constraint, where *a=c.abs()*,

FlightC.allInstances()-> forAll(c | a.preReserve() **implies** c.preReserve())

3.4 Correctness condition

We finally come to correctness. The correctness condition requires that an abstract operation can occur from an abstract state when the corresponding concrete operation can occur from a concrete state related to the abstract state by the retrieve relation R. The correctness condition requires, furthermore, that any state reached by performing the concrete operation is related by R to an abstract state reached by performing the abstract operation.

In a class diagram the effect of each operation $Op_i()$ is specified by attaching an OCL postcondition to $Op_i()$. In a postcondition, the expression can refer to values for each property of an object at two moments in time: the value of a property at the start of the operation and the value of a property upon completion of the operation. The value of a property in a postcondition is the value upon completion of the operation. In OCL, to refer to the value of a property at the start of the operation, the property name is decorated with the keyword "@pre".

In order to capture correctness in OCL, for each operation $Op_i()$ involved in the refinement we automatically generate a Boolean operation named *hasReturnedOpi(selfPre)*. This operation will evaluate *true* if the post condition of the operation $Op_i()$ is fulfilled. For example, given the specification of operation reserve() in figure 2, the following Boolean operations are generated,

context FlightA **def**: hasReturnedReserve(selfPre: FlightA): Boolean = self.reservedSeats = selfPre.reservedSeats + 1

context FlightC **def**: hasReturnedReserve(selfPre: FlightC): Boolean = self.seat -> exists (s | s.reserved **and** selfPre.seat -> exists(q | q.number = s.number **and not** q.reserved))

The expression *self* refers to the object that executed the operation (i.e. the after state), while the expression *selfPre* refers to the object that executes the operation, at the start of the operation (i.e. the before state). These operations

are automatically generated by applying minor adjustments to the postconditions attached to operations in the class diagram, such as the renaming of the occurrences of "`self.property_name@pre`" to "`selfPre.property_name`". Then, the Z correctness condition

$\forall c, c' \in CS\bullet$ pre Areserve $\wedge$ c Creserve c' $\Longrightarrow$ a Areserve a'

is emulated by the following OCL constraint, where a=c.abs() and `a_post=c_post.abs()`,

FlightC.allInstances() `->` forAll(c `|` FlightC.allInstances() `->` forAll(`c_post|` (a.preReserve()and `c_post` .hasReturnedReserve(c)) implies `a_post` .hasReturnedReserve(a)))

Observe that Z expressions of the form c COp_i c are encoded in OCL by creating expressions of the form "`c_post.hasReturnedCOpi(c)`". We use the postfix "`_post`" to emulate the decoration ' used in Z because OCL does not allow the use of nonalphabetical names. Also it was convenient for implementation reasons to overturn the order of decorated and undecorated variables (i.e., `c_post.hasReturnedOpi(c)` instead of `c.hasReturnedReserve(c_post)`.

4 Combining model checking, testing and semantic entailment for evaluating refinements

After creating the refinement conditions written in OCL we need to evaluate them using an OCL evaluator. In this section we describe our approach to evaluate OCL refinement conditions in an effective way.

The OCL is a Predicate Logic language. Two central concepts in Predicate Logic are semantic entailment (given a set of formulas Γ of predicate logic, determine whether $\Gamma \vdash \phi$ is valid) and model checking (given a formula ϕ of predicate logic and a matching model [4] M determine whether M $\models \phi$ holds). Semantic entailment matches well with software specification and validation; alas it is undecidable in general and would at least be intractable. On the other hand, a model M is a concrete instance of the system and all checks M $\models \phi$ have a definite answer: they either hold or do not. The problem with this approach is that a model M is not general enough; we are committing to instantiating several parameters which were left free in the requirements. From this point of view, semantic entailment is better because it allows a variety of models with a variety of different values for those parameters.

Looking for combining model checking and semantic entailment in a way which attempts to give us the advantages of both, Daniel Jackson in [14] presented the technique of micromodels of software. It consists in defining a finite bound on the size of models, and then checking whether all models M_i of that size satisfy the property under consideration (i.e., $M_i \models \phi$). This satisfaction checking has the tractability of model checking, while the fact that we range over a set of models

[4] Here the word "model" is used with the logic meaning which is subtly different from the modeling meaning.

allows us to consider different values of parameters gaining a considerable degree of generality:

- If we get a positive answer, we are somewhat confident that the property holds in all models. In this case, the answer is not conclusive, because there could be a larger model which fails the property, but nevertheless a positive answer gives us some confidence.
- If we get a negative answer, then we have found a model which violates the property. In that case, we have a conclusive answer, which is that the property does not hold.

From now on we will use the term micro-worlds instead of micromodels to avoid confusion between the logic and the modeling meaning of the term "model". Jackson's small scope hypothesis [14] states that negative answers already tend to occur in small worlds, boosting the confidence we may have in a positive answer.

4.1 Improving the micro-worlds by applying testing techniques

Even after defining a finite bound on the size of micro-worlds, we still might need to consider an infinite number of micro-worlds of that size. Thus, we should be able to select only a finite amount of representative micro-worlds.

To select useful micro-worlds we have to determine relevant values for the properties (attributes and multiplicities) of objects building up each micro-world. To achieve this requirement we developed an adaptation of a well-known testing strategy named categorypartition method [27]. Its main idea consists in dividing the domain into sub-domains or ranges that do not overlap each other and then to select values from each of these ranges. The category-partition method has been adapted to test UML models in [2]. Partitions provide a practical way to select representative values: for a property p and for each range G in the partition associated with p, the micro-world must contain at least one object o such that the value o.p belongs to G. For instance, the partitions `{{true}, {false}}` for the property "canceled" of class FlightC in the flights model, specifies that the micro-worlds should contain flights which are canceled and flights which are not canceled. The same kind of strategy is used for multiplicities of properties: if a property has a multiplicity of 0..300, a partition such as `{{0}, {1..299}, {300}}` is defined to ensure that the micro-worlds contain instances of this property with zero, 300 and an intermediate number of object. Figure 4 shows the partitions obtained for all properties of the flight model (partitions on the multiplicity of a property are denoted with the symbol sharp (`#`)). Default partitions based on the types of properties are automatically generated. However, if some values have a special meaning in the context of the refinement under validation, the user can modify the partitions. Additionally, the user might specify additional OCL constraints shaping the potential micro-world to be generated.

Partitions:

FlightC::canceled	`{true}, {false}`
FlightC::#seat	`{0}, {1..299}, {300}`
Seat::reserved	`{true}, {false}`
Seat::number	`{0}, {1..299}, {300}`

Additional Constraints:

context FlightA inv: self.reservedSeats <= self.capacity

Fig. 4. Partitions and constraints for the Flights

4.2 Tool support

The proposal presented in this paper is supported by ePlatero [10], an Eclipse plugin that we built on top of EMF [35] and GMF [35]. The tool allows us to create (or import) a class diagram containing a refinement, then the tool automatically generates the three refinement conditions (i.e. initialization, applicability and correctness). After that, the tool generates the micro-worlds and evaluates the refinement conditions on them. Figure 5 shows the architecture of the tool at a glance.

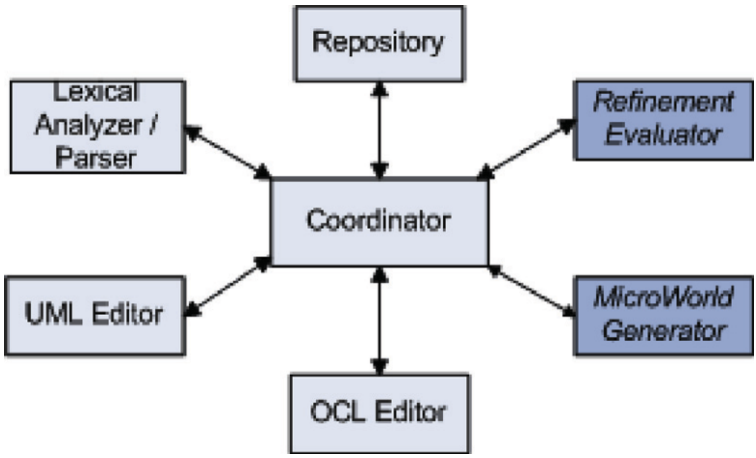

Fig. 5. Architecture of ePlatero

The Refinement Evaluator

The *Refinement Evaluator Component* implements the creation of refinement conditions by automatically applying the method described in this article. Figure 6 shows the main classes making up the component. The class *RefinementChecker* is a Singleton, it has exactly one instance. Its responsibility is to determine, for a given abstraction relationship, whether the refinement conditions hold or not. It collaborates with an instance of *RefinementConditionFactory* which has the responsibility to build and store the OCL file containing the OCL expressions to be

evaluated (i.e. initialization, applicability and correctness conditions).

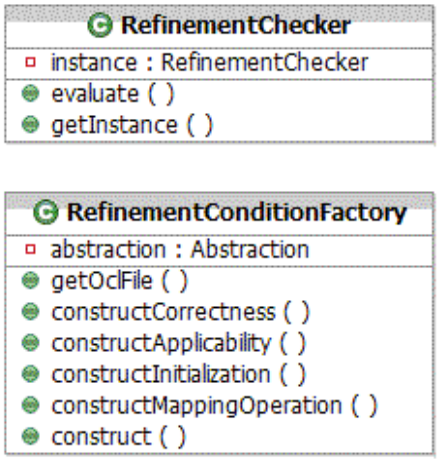

Fig. 6. Class diagram of the Refinement Evaluator Component

The MicroWorld Generator

The *MicroWorldGenerator* is the component having the responsibility to instantiate the set of objects making up snapshots of the system under evaluation. Its input consists of a set of OCL expressions determining additional domain restrictions, size restrictions and any further constraint that the developer might whish to include to shape the microworlds. It collaborates with the OCL evaluator in order to guarantee that the generated snapshots comply with the required OCL constraints. The *MicroWorldGenerator* uses the meta-model of Figure 7 to represent the notion of partition associated to properties. This meta-model distinguishes two types of partitions modeled by the classes *ValuePartition* and *MultiplicityPartition* that correspond to partitions for the value and the multiplicity of a property, respectively. For a *MultiplicityPartition*, each range is an integer range. For a *ValuePartition*, the type of ranges depends on the type of the property. Here the three primitive types that are defined in EMF are considered for the value of a property. Therefore, three types of ranges (StringRange, BooleanRange, IntegerRange) are modeled. The issue of building relevant micro-worlds cannot be resolved with a simple strategy such as creating all combinations of ranges for all properties of the input model. In general adequacy criteria are defined by specifying the properties that must be covered if the microworld is to be considered adequate with respect to the criterion [2]. Cost considerations and available resources often determine the selection of one criterion over another. Currently we offer two adequacy criteria: *OneRangeCombination* and *AllRangesCombination*. The first one is quite weak as it only ensures that each range of each property is covered at least once. The second is a lot stronger as it requires one object for each possible combination of ranges for all the properties of a class. Besides, the *Strategy design pattern* was used to implement such criteria, so that the tool can easily be extended to support additional

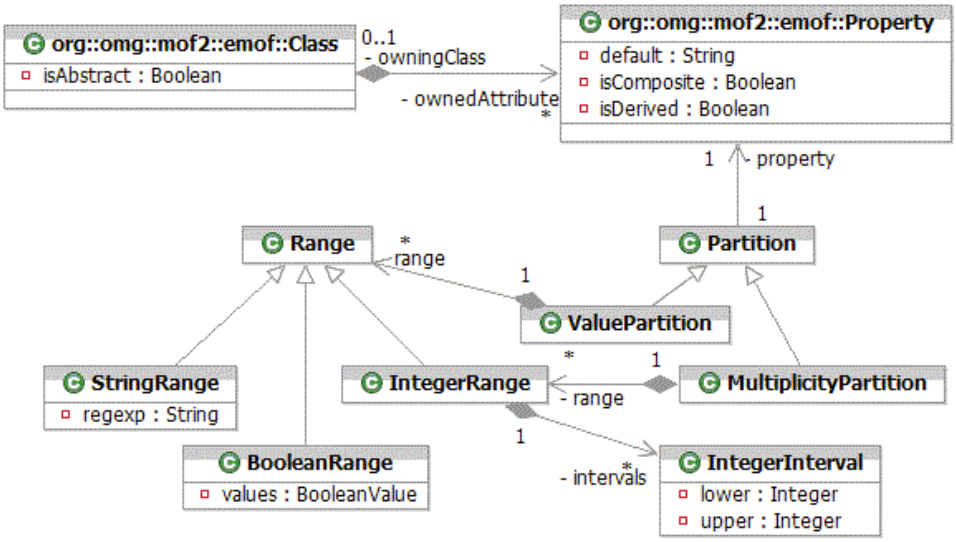

Fig. 7. Partition meta-model

criteria.

On the other hand, to be appropriate to analyze refinement relationships, the microworlds should satisfy the "duality property". Such property establishes that for each instance of a refined class at least one matching instance of the abstract class must exist (i.e., an instance related by the abstraction mapping). Therefore, we define partitions for each property of the refined classes only; then the values for properties of the abstract classes are automatically calculated by applying the abstraction function (i.e., the function abs() defined in the previous section). In this way we achieve two goals: first, the duality property holds trivially (i.e., by construction) and second, the amount of properties to be analyzed decreases significantly.

5 Experimental results

In this section we discuss the correctness and computational complexity of our approach by carrying out a comparison with the Alloy analyzer [13]. First, we translated the UML model in figure 2 to Alloy code, so that we were able to perform a formal evaluation of the refinement conditions by running the Alloy analyzer. We performed the translation to Alloy by applying the UML2Alloy tool [1] [5] and the proposal presented in [5] that shows how data refinement in Z can be automatically verified using the Alloy Analyzer.

Then, we made a comparison between the results reported by the Alloy analyzer and the results reported by ePlatero. The analysis was divided in two disjoint scenarios: evaluation of correct refinements and evaluation of incorrect refinements,

[5] We were forced to introduce minor adjustments on the generated Alloy code due to the fact that the UML2Alloy tool is not complete yet.

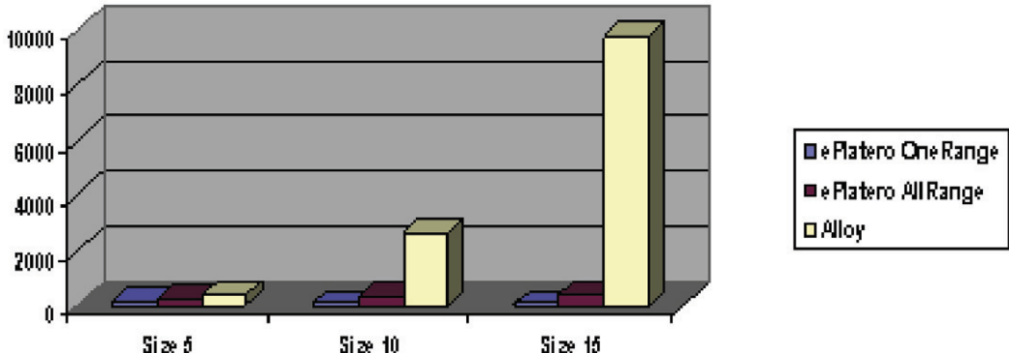

Table 1
Average evaluation costs according to the size of the micro-worlds

as it is presented in the following sections.

5.1 Correctness and complexity in the case of correct refinements

Due to the nature of both tools (i.e. counter example generation) the evaluations of refinement conditions are guaranteed to be correct in the case where the conditions actually hold (i.e. no false negatives are produced). Table 1 shows a comparison between ePlatero and Alloy with respect to the computation costs observed in the evaluation of the refinement in figure 2. The evaluation was repeated 100 times on an AMD Athlon 3000, for worlds of different sizes. The table shows the average costs (expressed in milliseconds).

5.2 Correctness and complexity in the case of incorrect refinements

To explore the scenario where refinement conditions do not hold, we introduce a mutation on the model in figure 2. Once again we carried out the evaluation 100 times. In this scenario we observed that both tools produced a number of incorrect responses (i.e., false positives). Table 2 shows the comparison between ePlatero and Alloy regarding the percentage of correct answers according to the size of the micro-worlds.

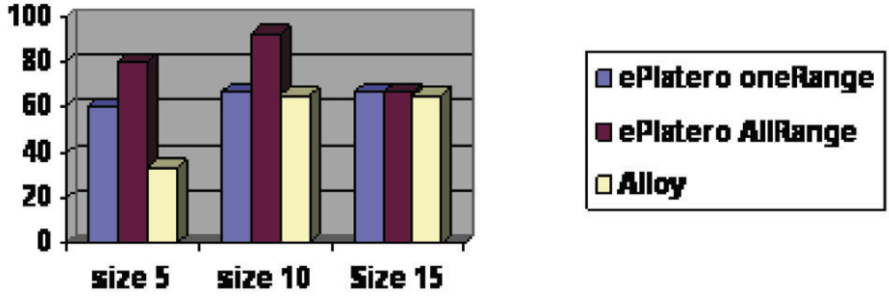

Table 2
Percentage of correct responses according to the size of the micro-world

We performed the same comparison between ePlatero and the Alloy analyzer with 10 UML models. These models were diverse in domain and size and they contain different kinds of refinement structures (see the report in [11]). Regarding correctness, the results were similar to the ones presented here, that is to say, both tools do not present significant differences. On the other hand, with respect to the computation costs, Alloy analyzer and ePlatero are similar when evaluating small models. But in the case of larger models, ePlatero presents a flexible micro-world generation strategy that allows us to find the most representative micro-worlds. On the one hand, we are able to take advantage of domain knowledge for improving the values in the partitions; consequently the tool will focus the analysis only on interesting values. And on the other hand new generation criteria (apart form the two built-in criteria) can be easily incorporated. The benefit of this flexibility was observed in the analysis of larger models, where significant improvement to evaluation costs was reported [11].

6 Related work

There are different alternatives to increase the robustness of the MDE refinement machinery by re-using a formal theory. One strategy consists in translating the core language used in MDE, i.e., UML/MOF, into a formal language, where properties are defined and analyzed. For example the works presented in [6] and [16], among others, belong to this group.

A second approach consists in applying the theory of graph transformation. As visual models can be seen as attributed graphs, application of graph transformation to specify model transformations has been a natural approach, giving rise to a number of proposals in recent years [38] [24] [18] [12].

Such proposals 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) models. However, such approaches are nonconstructive (i.e., they provide no feedback in terms of UML/MOF), they require expertise in reading and analyzing formal specifications and generally, properties that should be proved in the formal setting are too complex and undecidable.

Another alternative is to promote a formal definition of refinement, e.g., simulation, and emulate it in MDE terms. For example, Boiten and Bujorianu in [4] indirectly explore refinement through unification; Liu, Jifeng, Li and Chen in [22] define a set of refinement laws of UML models to capture the essential nature, principles and patterns of objectoriented design, which are consistent with the refinement definition; Paige and colleagues in [28] define refinement in terms of model consistency; Lano and colleagues in [19] describe a catalogue of UML refinement patterns which is a set of rules to systematically transform UML models to forms closer to Java code. Alexander Egyed in [9] presented a transformation-based consistency checking approach for consistent refinement, which is also lightweight since it does not require the use of third-party (formal) languages but instead integrates seamlessly into existing modeling languages. This approach considers only

the structural part of class diagrams because no behavior specification (e.g. pre and post conditions in OCL) is supported.

Following this later direction, in [29] we presented preliminary results on the formalization of step wise transformations of UML models by means of a set of heuristics for specifying and verifying refinement patterns that frequently occur in UML models. The present article describes a more general and fully automated approach. The main advantage of our proposal resides on the application of OCL as the only required formalism. In this way the definition and verification of transformations can be fully accomplished into a familiar development environment, without requiring developers with further knowledge and skills.

Finally, regarding decidability and tractability issues, the generation of micro-world is a pragmatic way to combine model checking and semantic entailment. Additionally, we improve the coverage of micro-world by the incorporation of testing technique. The idea of applying testing techniques to model transformations has also received increasing attention. Recent work by Baudry et al. [3] summarizes model transformation testing challenges.

7 Conclusion

Model-driven software engineering is seen as a promising approach to improve software quality and reduce production costs significantly. A major basis of such an approach is a usually domain-oriented modeling language which enables to abstract from implementation specific details and thus makes models easier to develop and analyze than the final implementation. In MDE models are supposed to be semi-automatically derived using model transformations, then the quality of these models will depend on the quality of model transformations. Each transformation step in the 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 validation task we considered how refinements between MOF/UML class diagrams can be checked by using tools that are fully understood by the MDE community. In particular, we show how the Object Constraint Language OCL [25] can be used to encode the standard simulation conditions. The proposed approach improves on existing approaches because it provides an efficient refinement evaluation mechanism that makes use of OCL as the only visible formalism, thus it integrates seamlessly into ordinary modeling environments. 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 proposed verification process is fully automatized. The software tool was integrated into ePlatero, however it is designed as an independent plugin so that it can be easily adapted to be attached to other modeling environments based on Eclipse. The computational complexity and correctness of the tool were empirically

evaluated in a number of case studies [11] and such properties were observed to be acceptable (and even improved) with respect to the ones of the Alloy Analyzer which is a well accepted and mature formal tool.

We believe that 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. This is an important step towards fully verified model transformations, which are necessary to guarantee the correctness of the generated implementations of abstract models.

References

[1] Anastasakis, K., B. Bordbar, G. Georg, and I. Ray, *UML2Alloy: A Challenging Model Transformation*, ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems, LNCS **4735** (2007), Springer, 436-450.

[2] Andrews, A., R. France, R. Ghosh, and G. Craig, Test adequacy criteria for UML design models, Software Testing, Verification and Reliability, **13(2)** (2003), 95-127.

[3] Baudry B., T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon, *Model Transformation Testing Challenges*, Proceedings of IMDT workshop in conjunction with ECMDA (2006).

[4] Boiten E.A., and M.C. Bujorianu, *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).

[5] Bolton, Christie, *Using the alloy analyzer to verify data refinement Z*, Proceedings of the REFINE 2005 Workshop, Electronic Notes in Theoretical Computer Science, **137**(2005), 23-44.

[6] Davies J., and C. Crichton, *Concurrency and Refinement in the Unified Modeling Language*, Electronic Notes in Theoretical Computer Science, Elsevier.(2002).

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

[8] Dijkstra, and W. Edsger, " A Discipline of Programming," Prentice Hall Series in Automatic Computation, 1976.

[9] Egyed Alexander, *Consistent Adaptation and Evolution of Class Diagram during Refinement*, 7th Int. Conf. on Fundamental Approaches to Software Engineering FASE (2004).

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

[11] Garcia, D., and C. Pons, *Refinement Pattern Catalog. LIFIA Technical Report.*, http://sol.info.unlp.edu.ar/eclipse/refinementsCatalog.pdf, 2007.

[12] Giese, H., S. Glesner, J. Leitner, W. Schafer, and R. Wagner, *Towards Verified Model Transformations*, MODEVA Workshop at MODELS, 2006.

[13] Jackson, D, Alloy Analyzer website http://alloy.mit.edu/, 2007.

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

[15] Jouault F., and I. Kurtev, *Transforming Models with ATL*,Model Transformation in Practice Workshop at MODELS Conference (2005).

[16] Kim, S., and D. Carrington,*Formalizing the UML Class Diagrams using Object-Z*, proceedings UML99 Conference, Lecture Notes in Computer Sciencie **1723** (1999).

[17] Kleppe, Anneke, J. Warmer, and W. Bast, "MDA Explained: The Model Driven Architecture: Practice and Promise,"Addison-Wesley Longman Publ. Co., Boston, USA (2003).

[18] Küster, J.M.,*Definition and Validation of Model Transformations*, Journal on Software and Systems Modeling, **5, Springer**(2006), 233-259.

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

[20] Lawley M.,and J. Steel , *Practical Declarative Model Transformation with TefKat*, Model Transformation in Practice Workshop at MODELS Conference (2005).

[21] Le Traon, Yves, Baudry, Benoit and Mottu, Jean-Marie, *Reusable MDA Components: A Testing-for-Trust Approach*, MoDELS 9th International Conference, Lecture Notes in Computer Science **4199**(2006), 645-659.

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

[23] Meta Object Facility (MOF) 2.0 Core Specification. OMG, 2005.

[24] Narayanan, A., G. Karsai, *Towards verifying model transformations*, In: 5th International Workshop on Graph Transformations and Visual Modeling Techniques, Electronic Notes in Theoretical Computer Sience (2006), 185-194.

[25] Object Constraint Language OCL 2.0. OMG Final Adopted Specification. ptc/03-10-14, 2003.

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

[27] Ostrand T.J., and M.J. Balcer, *The category partition method for specifying and generating functional tests*, Communications of the ACM **31(6)**(1988), 676-686.

[28] Paige, R., D. Kolovos, and F. Polack, *Refinement via Consistency Checking in MDD*,Procs. of REFINE'2005. Electronic Notes in Theoretical Computer Science **137**(2005), 151-161.

[29] Pons, C.,and D. Garcia, *An OCL-based Technique for Specifying and Verifying Refinement-oriented Transformations in MDE*,Model Driven Engineering Languages and Systems, 9th International Conference, Lecture Notes in Computer Science **4199**(2006), 645-659.

[30] Query/View/Transformations - OMG Adopted Specification. Document ptc/05-11-01, http://www.omg.org, 2005.

[31] Smith, Graeme and John Derrick, *Verifying data refinements using a model checker*,In Journal Formal Aspects of Computing, Springer London **18:3**(2006).

[32] Smith, G., and K. Winter,*Proving temporal properties of Z specificatons using abstraction.*,International conference of Z and B users **2651 of LNCS, Springer**(2003), 260-279.

[33] Spivey, J. M, "The Z Reference Manual,"Prentice Hall International (UK) (1992).

[34] Stahl, M. Voelter, " Model Driven Software Development," John Wiley (2006).

[35] The Eclipse Project: Eclipse Modeling Framework EMF, http://www.eclipse.org/modeling/emf/, Graphical Modeling Framework GMF, http://www.eclipse.org/modeling/gmf/,

[36] UML 2.0, The Unified Modeling Language Infrastructure version 2.0 - OMG Final Adopted Specification, formal/2005-07-04, http://www.omg.org, 2005.

[37] UML 2.0, The Unified Modeling Language Superstructure version 2.0 - OMG Final Adopted Specification, formal/2005-07-04, http://www.omg.org, 2005.

[38] Varro, D., and A. Pataricza, *Automated formal verification of model transformations*, CSDUML: Critical Systems Development in UML, Proceedings of the UML'03 Workshop (2003).