Generating operation specifications from UML class diagrams: A model transformation approach

Manoli Albert, Jordi Cabot, Cristina Gómez, Vicente Pelechano

Generating Operation Specifications from UML Class Diagrams: A Model Transformation Approach

Manoli Albert[1], Jordi Cabot[2], Cristina Gómez[3], Vicente Pelechano[1]

[1]Departmento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia
Camino de Vera s/n 46022 Valencia (Spain)
{malbert,pele}@dsic.upv.es

[2]INRIA-École des Mines de Nantes
4, rue Alfred Kastler, B.P. 20722 - F-44307 NANTES Cedex 3 (France)
Jordi.cabot@inria.fr

[3]Departament d'Enginyeria de Serveis i Sistemes d'Informació, Universitat Politècnica de Catalunya
Campus Nord, Edif. Omega, Jordi Girona 1-3, 08034 Barcelona (Spain)
cristina@essi.upc.edu

Abstract.

One of the more tedious and complex tasks during the specification of conceptual schemas (CSs) is modeling the operations that define the system behavior. This paper aims to simplify this task by providing a method that automatically generates a set of basic operations that complement the static aspects of the CS and suffice to perform all typical life-cycle create/update/delete changes on the population of the elements of the CS. Our method guarantees that the generated operations are executable, i.e. their executions produce a consistent state wrt the most typical structural constraints that can be defined in CSs (e.g. multiplicity constraints). In particular, our method takes as input a CS expressed as a Unified Modeling Language (UML) class diagram (optionally defined using a profile to enrich the specification of associations) and generates an extended version of the CS that includes all necessary operations to start operating the system. If desired, these basic operations can be later used as building blocks for creating more complex ones. We show the formalization and implementation of our method by means of model-to-model transformations. Our approach is particularly relevant in the context of Model Driven Development approaches.

Keywords: Behavior schema, action, operation, class diagram, UML, model-to-model transformation

1. Introduction

Current Model Driven Development (MDD) and Model Driven Architecture (MDA) approaches propose applying a chain of model-to-model (M2M) transformations to (automatically) derive the final implementation of the system from its initial conceptual schema (CS).

One of the most tedious and complex tasks during the definition of CSs is the specification of all operations that describe the behavior of the system. In fact, in order to be completely functional, designers must provide a detailed specification of the change effect of each operation on the system state. The operation specification must

take into account the constraints specified in the CS to make sure they are fulfilled after the operation is executed. This specification can be provided in different languages, such as: natural language, imperative action-based languages [1] or as declarative contracts expressed, for instance, with the Object Constraint Language (OCL) [2]. In any case, operation definitions are linked to the specification of the CS itself, usually performed by means of a graphical modeling language like UML, Entity-Relationship (ER) [3] or Object Role Modeling (ORM) [4]. In this paper we will use UML as the standard notation for drawing CSs [5] and OCL and an action-semantics based imperative language for the operation specifications.

Even though the detailed definition of the system behavior is a prerequisite for MDD, results of recent surveys about the use of UML among practitioners (e.g. [6]) clearly show that most practitioners only focus on the static aspects of class diagrams in their day-to-day practice and ignore all other modeling aspects and diagrams. This hampers the application of MDD approaches in practice. Therefore, it is clear that any technique that can facilitate the definition of such aspects would be really helpful and could improve the adoption level of MDD techniques among the software engineering community.

In this sense, this paper provides a method to automate the generation of a basic behavior specification for the system modeled in the UML class diagram. This improves the productivity and quality of the designers' job since writing a behavior specification is a very time-consuming and error-prone task. Although our method can be useful in any application domain, our method is specially useful for domains that make an intensive use of data-manipulation operations. The basic behavior specification generated by our method consists in a set of well-formed set of operations that suffice to cover most of the required system's behavior. More specifically, the generated operations allow designers to perform all required life-cycle change events (create/delete/update) on the population/value of the different model elements of the class diagram. The number and effect of the operations are determined based on the domain knowledge contained in the static structure of the class diagram.

The work reported here extends our previous work [7] in several directions. First, the input of our method is now a UML class diagram that may contain enriched association definitions. These additional properties for associations (introduced in [8]) convey more information on the association semantics. This additional knowledge helps us to obtain a set of operations closer to the one that the designer would manually specify. A second contribution is that in this work we focus on the generation of strongly executable operations. That is, we provide a more complete operation specification that ensures that the operation execution *always* maintains the system consistency regarding the structural constraints and dependencies between the elements of the class diagram (e.g. multiplicity constraints). For these constraints, our operation definition guarantees that for any invocation where the precondition is satisfied the operation evolves the system state to a new consistent state. Instead, our previous work focused on a weaker executability property that only guaranteed that there was at least a chance of obtaining a consistent state. Additionally, we formalize our method as a M2M endogenous transformation using the ATLAS Transformation Language (ATL) [9] and implement it in the Eclipse-based *MOSKitt* [10] open-source CASE tool.

Our method advances the current state-of-the-art in the area of behavior specification generation methods such as [11-27] in several aspects. First, our method deals with more expressive input models (e.g. including enriched associations and abstract classes) which has a direct impact on the number and the completeness and richness of the generated operations. Second, our method mixes two different strategies to avoid inconsistencies: checking (by adding preconditions) and maintenance (by adding actions that repair the inconsistency). Choosing one or the other depending on the operation semantics improves the user experience. Only [13, 14] also consider both strategies but only for a reduced set of properties (for instance, they ignore properties as important as the minimum and maximum multiplicities of associations). Moreover, many of the reviewed methods need a manual specification of the operations [11-15, 20, 21] or do not provide tool support and the ones that provide an automatic method with tool support (as it is our case) generate less operations and simpler ones than those generated by our method as reviewed in detail in Section 7.

The remainder of the paper is structured as follows. Section 2 presents an overview of our method. Section 3 introduces some basic concepts regarding structural and behavioral aspects of class diagrams. Section 4 explains in detail each step of the method. In Section 5 the M2M transformation using the ATL language is defined and introduces the tool support. Section 6 presents the results of applying our method to different scenarios. Finally, Section 7 reviews the related work and Section 8 draws some conclusions and describes further work.

2. Method Overview

Our method takes as input a UML-based class diagram with only the static aspects specified and returns as output a class diagram where classes have been extended with the operations required to modify the system state. The number and specification of these operations are deduced from the properties and dependencies between the structural aspects of the class diagram.

The main particularity of our method is that the specification of each operation includes the functionality that is necessary to guarantee the fulfillment of the structural properties of the elements of the class diagram (as multiplicity constraints, disjointness and so on; see Section 3). These properties state conditions that must be satisfied by the system state at run-time. All generated operations always leave the system in a consistent state wrt these conditions at the end of the operation execution (i.e. the operations are *strongly executable*).

Our method can be split up into three main steps (Fig. 1):

1. *Identification of operations.* The method identifies which operations should be defined to carry out the necessary modifications on the population or values of the different elements of the class diagram, together with the classes where these operations have to be attached to.
2. *Specification of the operation bodies.* The body of each operation includes the logic of the operation (i.e. basic actions as insertion of a new object, deletion of a link, …) plus the additional functionality (i.e. other actions and/or preconditions) to guarantee that the structural properties that may be affected by the operation are not violated during its execution. This added functionality is needed to satisfy possible dependencies between the actions in the operation since some actions may require

the presence of other actions in the same operation in order to be able to leave the system in a consistent state.

3. *Specification of the operation signatures*. The signature of each operation is derived from the actions included in the operation body. The final signature of each operation is decided in this last step when all dependencies between the actions in the operation are satisfied.

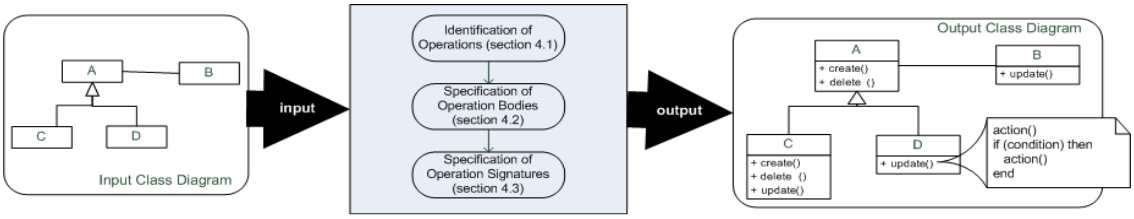

Figure 1. Overall view of the method

We assume in this paper that the initial UML CS defined by the designer is consistent. By consistent we mean that it is *strongly satisfiable*, i.e. it is possible to create valid instances of the CS. An instance is valid if it satisfies all the constraints of the CS. Otherwise, the designer must, first, fix the CS before applying our method. There are several tools available to ensure the consistency of a UML CS, as for instance [28].

3. Basic Concepts

This section briefly reviews the basic terminology and definitions used in this paper regarding the specification of the structural and behavioral aspects of a system. In our approach both aspects are described by means of a class diagram expressed in UML 2, though, some elements of the class diagram include slight extensions wrt the standard elements in the UML metamodel. In particular, we introduce a set of new properties that allow associations to be characterized in a more complete and clear way than in the UML proposal. A UML profile for this extension has been defined in [29].

Section 3.1 describes the structural aspects of our UML-based class diagrams and section 3.2 focuses on the behavioral aspects.

3.1. Structural View

The constructs that our method considers to specify the structural aspects of a system are classes, attributes, binary[1] associations and generalizations. For the sake of clarity, association classes are not considered[2].

Classes may be *concrete* or *abstract* and attributes may be defined as *readOnly*, *derived* and *isNotNull* (the multiplicity is exactly one) [31]. For the sake of simplicity we assume that all attributes have a maximum multiplicity of 1[3]. The function *isAbstract(Cl)* returns *true* when the *Cl* class is abstract and *false* otherwise. The functions *isReadOnly(at), isDerived(at)* and *isNotNull(at)* are similarly defined.

[1] N-ary associations can be easily expressed in terms of a set of binary ones plus additional constraints [30].

[2] An association class could be represented as a regular class with *n* associations one for each participant plus a constraint restricting that no instance of the class can be related with the same exact set of participants.

[3] Multi-valued attributes can be represented and treated as an association between the class owning the attribute and the corresponding data type.

Association ends (also known as *roles*) are the endpoints of associations. Each association end connects the association to its participant class. Standard UML 2 properties upon association ends (considered in this paper) are *maximum* and *minimum* multiplicities, *derived* and *navigability* properties. The functions $max(p_1; As)$ and $min(p_1; As)$ specify the maximum and the minimum multiplicity of the *As* association between classes Cl_1 (playing the p_1 role) and Cl_2 (playing the p_2 role) and $navigability(p_1; As)$ and $derived(p_1; As)$ define if the association end p_1 is navigable and derived, respectively.

A *generalization set g*, denoted by $Gen(Cl;Cl_1,...,Cl_n)$ between a more general class *Cl* (superclass) and a set of more specific classes $Cl_1,...,Cl_n$ (subclasses) may be *disjoint* and *complete*. Functions *isDisjoint(g)* and *isCovering(g)* return *true* when *g* is disjoint and complete, respectively.

Our method also considers a more advanced characterization of associations since the characterization provided in the literature for the association concept (see [1, 32-34]) experiences some drawbacks that make the use and interpretation of this construct ambiguous. Several works [35-41] have tried to propose alternative semantics. In this method the particular interpretation of the association concept introduced in [8] is taken into account. This interpretation allows this method to extract more precise knowledge from the class diagram in order to improve the specification of the generated operations to maintain the association population at run-time. The additional association end properties defined in [8] and used in this paper are:

- **Changeability (***changeability(p_1; As)***)**: Specifies whether links can be created or deleted after the initialization of objects of the Cl_2 class. Possible values are:
  - *unrestricted*: no restrictions on creation and destruction of links. This is graphically represented annotating the role with the <<+,->> stereotype.
  - *addOnly:* links cannot be deleted after the participating objects of the Cl_2 class have been initialized. Represented with the <<+>> stereotype
  - *removeOnly*: new links cannot be created after the objects of the Cl_2 class have been initialized. Represented with the <<->> stereotype
  - *readOnly:* links can neither be deleted nor inserted after the objects of the Cl_2 that participate in the links have been initialized

The default value of the *changeability* property is *unresticted*. The property in UML 2 [1] that is closest to changeability is the boolean *isReadOnly* property of association ends. This property maps to our *readOnly* value (for *true* values) and to our *unrestricted* value (for *false* values).

- **Delete Propagation (***delpropagation(p_1; As)***):** Indicates which actions must be performed when an object of the Cl_2 class is deleted. The possible values are:
  - *restrictive*: the object of the Cl_2 class to be deleted cannot be deleted if it has links (an exception is raised if an attempt is made); otherwise, it is deleted. The restrictive value is depicted by means of the <<RT>> stereotype.
  - *cascade*: links of the object of the Cl_2 class to be deleted and its linked objects must also be deleted. Cascade roles are annotated with the <<CC>> stereotype.
  - *link* (default value): links of the object of the Cl_2 class to be deleted must also be deleted (but not its linked objects). No stereotype is needed in this case.

We would like to remark that the introduced properties are not orthogonal, i.e. there exist dependencies among the properties. In [29] those dependencies are analyzed. We assume that class diagrams satisfy all these dependencies.

For the sake of simplicity, our method does not consider, in the current version, other UML properties as subsetting and redefinition for attributes and ordering, redefinition, subsetting and uniqueness for association ends. Instead, the method may be applied to other constructs (as for instance, compositions and aggregations) that may be defined as a combination of the constructs and properties defined above. For example, a composition is dealt by our method as an association between the composite and its parts where the minimum and maximum multiplicity of the composite class role is one and non-navigable, the part class role is navigable and the value of the delete propagation property is cascade.

As a running example throughout the rest of the paper, we use the class diagram shown in Fig. 2. This class diagram represents a disjoint and complete generalization between the *Document* abstract class and the *InternalDocument* and *Publication* classes. The *name* attribute of the *Document* and *Conference* classes and the *number* attribute of the *ConferenceEdition* class are read only and not null. Moreover, the *acceptanceRatio* attribute of the *ConferenceEdition* class is derived since its value may be calculated as a percentage between the number of papers submitted (*numberOfSubmissions*) and the number of papers published (*participant* role). The changeability of the *in* role (*Publishes* association) is defined as *addOnly* since a publication cannot ever delete its link with the conference edition in which it was published; however, a publication not yet published can be linked to a conference at any time. Besides, a conference edition cannot be deleted when it participates in a link associated to a publication, so delete propagation at the *participant* role is marked as *restrictive*. The non-standard properties for associations are defined using the UML profile proposed in [29]. We use the implicit notation [31] to represent the navigability of associations. That is, single arrows indicate one-way navigable associations and no arrows two-way navigable associations.

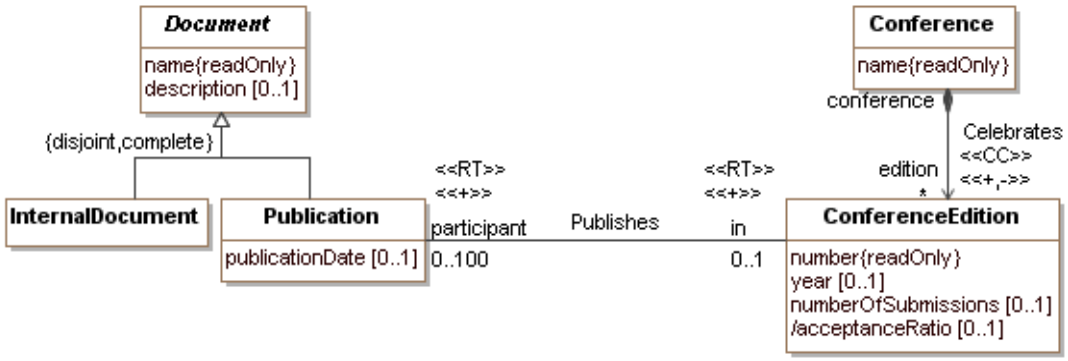

Figure 2. Class diagram used as a running example

3.2. Behavioral View

This section introduces some preliminary concepts to specify the behavioral aspects of a system. These concepts will be used by our method when creating the specification of operations for the input static class diagram.

In our case, the operation effect specification is defined in an imperative way. For each operation the set of basic actions that are applied on the system state when the operation is executed is explicitly defined. We use the term *basic action* (also known as structural event) to refer to an atomic change on the population or value of the system state.

3.2.1. Basic Actions

The list of basic actions that our method considers is the following:
- *iCl(x)*: creates a new *x* object in the *Cl* class
- *dCl(x)*: deletes the *x* object from the *Cl* class
- *uAt(x,v)*: updates the *At* attribute of the object *x* with the *v* value
- *iAs(x,y)*: creates a new link for the *As* association with *x* and *y* as participant objects
- *dAs(x,y)*: deletes the link <*x,y*> from the *As* association
- $sCl_pCl_c(x)$: specializes the object *x* of Cl_p superclass to the Cl_c direct subclass. The action is only applicable if *x* is an instance of Cl_p and not of Cl_c
- $gCl_cCl_p(x)$: generalizes the object *x* of Cl_c subclass to the Cl_p direct superclass. This action is applicable if *x* is an instance of Cl_c and not of its subclasses.

We also predefine two frequent compound actions that facilitate the definition of our method:
- $uAs(x,y_1,y_2)$: replaces the $<x,y_1>$ link in *As* with a new $<x,y_2>$ link
- $uCl_{c1}Cl_{c2}(x)$: moves *x* from Cl_{c1} to Cl_{c2} (i.e. generalizes *x* to the supertype and specializes it as a new instance of Cl_2).

Our list of basic actions is a more fine-grained version (i.e. more elementary) that those proposed in the UML Action Semantics [1] (e.g. the UML reclassify action is treated as a sequence of individual generalization and specialization actions). This permits a more detailed reasoning when generating the operations. Nevertheless, a correspondence between the basic actions that our method considers and the ones provided by UML standard is straightforward, see [42, p. 32] for mapping table between the two.

At the syntax level, since the UML does not predefine any concrete syntax for expressing the actions, we have defined our own textual representation.

3.2.2. Operations

An operation is a behavioral feature of a class that specifies the name, type, parameters, and constraints (preconditions) for invoking an associated behavior. As we have said above, the operation behavior is specified by explicitly listing the set of actions that are executed when calling the operation. We assume that the behavior of the operations is transactional; it means that the set of actions that define the operation is treated as one atomic execution unit. In this way, all the participating actions should either succeed or fail, but if they fail the previous state before the execution must be recovered.

An example of an operation with its specified effect for our running example is the operation ConferenceEdition::createConferenceEdition:

```
ConferenceEdition::createConferenceEdition(v:Integer,
conf:Conference){
    iConferenceEdition(x);
```

```
uNumber(x,v);
iCelebrates(x, conf); }
```

In fact, this is one of the operations that are automatically generated by our method. Note that the operation initializes the *number* and the *Celebrates* properties; otherwise the operation would not be executable, as we will also explain in Section 4.2.

4. Generating Operation Specifications from UML Class Diagrams

This section presents our method for deriving behavioral specifications from static models. Our method first analyzes the class diagram and extracts relevant information from the properties of the model elements and from the relationships between them. Then, this information is used to determine the system operations that are necessary to carry out all typical modifications on the population or values of different elements of the class diagram (Section 4.1). Once these operations have been identified, our method generates their body, including the definition of the maintenance strategies (e.g. in the form of additional required actions or preconditions that must appear together with a given action type) that must be added to the operations' behaviors to ensure that all of them satisfy the strong executability correctness property wrt to the structural properties considered (Section 4.2). Finally, our method defines the signature of each operation (Section 4.3).

4.1. Identifying Operations

This section introduces the rules that allow our method to identify the set of operations that suffice to provide basic insert/update/delete functionality for the different elements of the class diagram. We take into account the properties of each model element to avoid creating unnecessary operations.

Each type of element of the class diagram (class, attribute, association and generalization set) has an associated rule. Each rule determines the set of operations that must be defined for elements of that type, depending on the properties of each element. For example, the rule that determines which operations should be defined for classes, determines that a *createCl* operation is only generated for non-abstract classes that are not superclasses of covering generalizations.

Our method uses the following rules for each element of the input class diagram to identify the operations of each class:

Rule 1 (Classes). For each *Cl* class in the input model, **if [***Cl is not isAbstract(Cl) and is not superclass of a covering generalization***]**, **generate** a *createCl* operation in *Cl*. Additionally, **if [***Cl does not participate in an* $As(p_1{:}Cl;p_2{:}Cl_2)$ *association so that* $delpropagation(p_2;As) = restrictive$ *and* $min(p_2;As) > 0$**]**, **generate** a *deleteCl* operation in *Cl.*

The rationale of the rule is that object creations may only be performed in concrete classes that are not superclasses of covering generalizations. Therefore, only for those classes, a *createCl* operation should be created. For other classes (e.g. abstract classes), creation of class objects is performed as a consequence of creation of objects in one of

its subclasses. The second part of the rule ensures that deletion of objects of a class is only possible for those classes not at the opposite end of an association marked as *restrictive* and with the minimum multiplicity greater than 0. Instances of those classes can only be deleted as part of the deletion of instances of that association or instances of the opposite class.

For the running example, this rule adds eight new operations. Fig. 3 shows the application of the rule for all the classes of the example (the figure focuses only on the elements and properties involved in this rule). Note that for the *Document* class no operations have been generated in the output class diagram, since it is an abstract class.

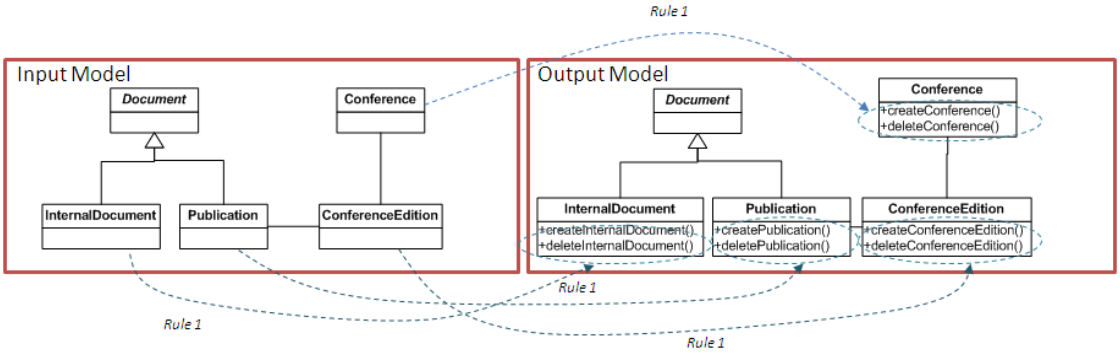

Figure 3. Rule 1 applied to classes of the running example

Rule 2 (Attributes). For each *At* attribute of *Cl* in the input model, **if [***At is not isDerived(At) and not isReadOnly(At)***]**, **generate** an *updateAt* operation in *Cl.*

In this rule, the rationale is that an operation to update an attribute value must be generated in the class that owns the attribute if the attribute is not derived and not read only. Attributes that are derived or read only cannot be modified by the designer, so none of the operations should be generated for this purpose.

For the running example, this rule would generate four new operations (see Fig. 4).

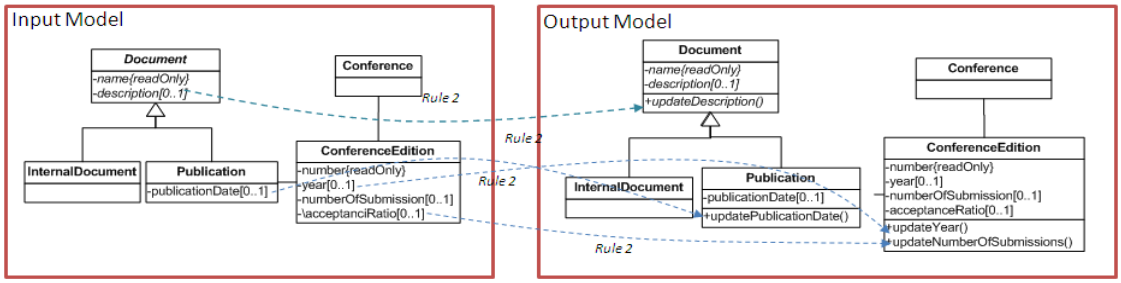

Figure 4. Rule 2 applied to attributes of the running example

---

Rule 3 (Associations). **For each** *AsE* association end of an *As* association and Cl_2 class (being Cl_1 the opposite class), **if [***As is not isDerived(As) and navigability(p_2;As)***]** and:

- **[***changeability(p_2;As) = addOnly or unrestricted and min(p_2;As) ≠ max(p_2;As)***]**, **generate** a *createAs* operation in Cl_1
- **[***changeability(p_2;As) = removeOnly or unrestricted and min(p_2;As) ≠ max(p_2;As)***]**, **generate** a *deleteAs* operation in Cl_1
- **[***changeability(p_2;As) = unrestricted***]**, **generate** an *updateAs* operation in Cl_1

The rationale of the rule is that when an association is not derived and a class that participates in the association can navigate to the class at the opposite end (navigability is *true* at the opposite end) operations to create, delete and update links are generated in that class depending on the values of the *changeability* and *multiplicity* properties at the opposite end. Specifically, if *changeability* of an association end is *unrestricted* then objects of the class at the opposite end may update their links, so the operation to update the links must be generated. Moreover, an object may add new links if its *changeability* is *addOnly* or *unrestricted* and the *maximum* and the *minimum* multiplicity is not the same. In this case, the operation to create the links must be generated. In the same way, an object may remove its links if its *changeability* is *removeOnly* or *unrestricted* and the *maximum* and the *minimum* multiplicity is not the same. The operation to delete the links must be generated.

For the running example, this rule would generate five new operations (see Fig. 5).

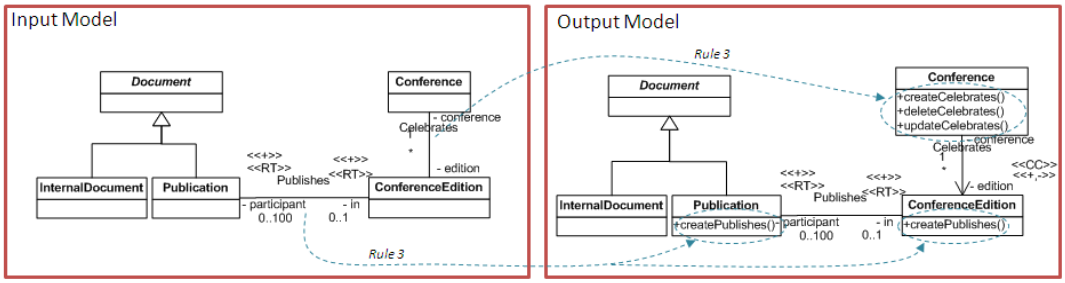

Figure 5. Rule 3 applied to associations of the running example

Note that the *createPublishes* operation is replicated in two classes (*Publication* and *ConferenceEdition* classes) due to the navigability and changeability values of the *Publishes* association. This allows creating an instance of the association from both classes.

Rule 4 (Generalizations). **For each** $g=Gen(Cl_p;Cl_{s1},...,Cl_{sn})$ generalization set, **generate** an $updateCl_{si}Cl_{sj}$ operation in all Cl_{si} subclasses (i,j=1,..,n, i≠j). Additionally, **if [***g* is not isCovering(*g*) or not isDisjoint(*g*)**]**, **generate** a $specializeCl_pCl_{si}$ operation in Cl_p (∀ i=1,..,n) and **generate** a $generalizeCl_{si}Cl_p$ operation in all Cl_{si} subclasses (∀ i=1,..,n).

The rationale of this rule is that, for generalization sets, operations for moving instances from each subclass of the generalization set to the rest of the subclasses must be always generated. The second part of the rule states that operations for specializing and generalizing instances must be only generated for generalization sets that are not covering or not disjoint. A generalization set that is covering and disjoint does not require operations for specializing or generalizing instances.

For the example, this rule would generate two new operations (see Fig. 6). Note that only update operations have been generated since the generalization set of the example is covering and disjoint.

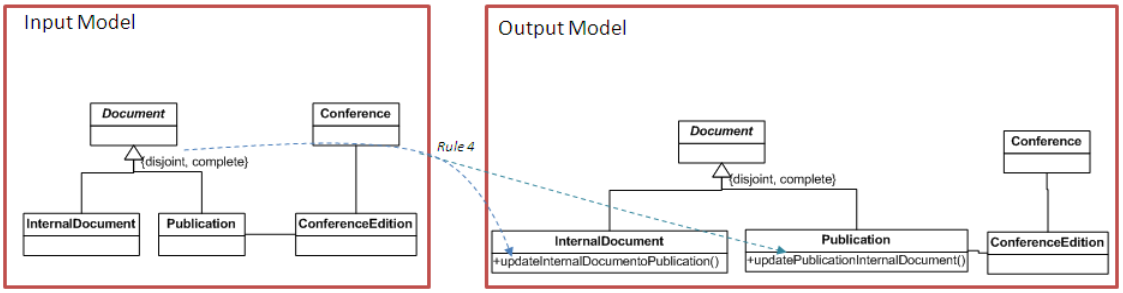

Figure 6. Rule 4 applied to the generalization set of the running example

4.2. Specifying Operation Bodies

Next step consists in generating the imperative specification body for each operation identified in the previous step. Initially, the body of those operations just contains the single basic action that implements the semantics of the operation (see Table 1).

Table 1. Actions Types contained in Operations

Operation	Action
createCl at *Cl*	*iCl*
deleteCl at *Cl*	*dCl*
updateAt at *Cl*	*uAt*
createAs at Cl_1 and Cl_2 (where Cl_1 and Cl_2 are the participant classes in *As)*	*iAs*
deleteAs at Cl_1 and Cl_2 (where Cl_1 and Cl_2 are the participant classes in *As)*	*dAs*
updateAs at Cl_1 and Cl_2 (where Cl_1 and Cl_2 are the participant classes in *As)*	*uAs*
specializeCl_pCl_{si} at Cl_p (where *Gen*(Cl_p,Cl_{s1}, ...,Cl_{sn})	sCl_pCl_{si}
generalize$Cl_{si}Cl_p$ at Cl_{si} (where *Gen*(Cl_p,Cl_{s1}, ...,Cl_{sn})	$gCl_{si}Cl_p$
update$Cl_{si}Cl_{sj}$ at Cl_{si} (where *Gen*(Cl_p,Cl_{s1}, ...,Cl_{sn})	$uCl_{si}Cl_{sj}$

However, in general, this is not enough to guarantee that the operation execution will respect the integrity constraints defined in the model. Many operations will need to add new functionality to guarantee that the constraints that may be affected by the operations are not violated during its execution. This added functionality is needed to satisfy possible dependencies between the actions in the operation since some actions may require the presence of other actions in the same operation in order to be able to leave the system in a consistent state. For instance, consider the *createConference* operation (which contains the *iConference* basic action according to Table 1) of the `Conference` class (shown in Fig. 3). The creation of a conference requires the specification of a value for the *name* attribute since it cannot be null (*isNotNull(name)* = *false* according to Figure 2). To avoid the violation of the *isNotNull* property the *uName* action should also be included within the *createConference* operation.

In some other cases, this new functionality will come in the form of preconditions for the operation that prevent the execution of the operation on those states in which the changes performed by the operation would leave the system in an inconsistent state. For instance, consider the *deleteConferenceEdition* operation (which contains the *dConferenceEdition* basic action according to Table 1) of the `ConferenceEdition`

class (shown in Fig. 3). The deletion of a conference edition when it has a publication associated to it may not be performed since the *delete propagation* value of the role *participant* is *restrictive*. To prevent this deletion a precondition to guarantee that the conference edition has not publications is required.

The specification of the operation effect (i.e. the operation body) for an operation *op* initialized with a basic action *ac* follows these steps:

1. Adding to *op* the $ac_1..ac_n$ actions or/and $pr_1 \ldots pr_n$ preconditions that are necessary to guarantee that the structural properties (or constraints) that may affected by the action *ac* are not violated during its execution (see Section 4.2.1).
2. Applying recursively step 1 to the new actions (ac_1 and ac_2, ac_3,…) added to *op* until no more actions or preconditions are required. This is necessary in order to satisfy the dependencies of the new actions (ac_1 and ac_2, ac_3,…) within the operation. When an action ac_i in *op* has as required action an action ac_j that is already part of *op*, the required action is considered to be satisfied and ac_j is not added to *op* again. In some uncommon scenarios this recursive process to generate the operation effect may not terminate. Our method identifies these scenarios and requires the designer to take part in the process to avoid an infinite loop.

During the previous steps, the following considerations apply:

- When *ac* requires an iCl_{c2} or dCl_{c1} action, being Cl_{c1} and Cl_{c2} subclasses of the same generalization, and *op* already includes the $uCl_{c1}Cl_{c2}$ action, the required maintenance action is also considered to be satisfied (since $uCl_{c1}Cl_{c2}$ performs the changes of the iCl_{c2} action and the dCl_{c1} action). The same reasoning has to be applied to the *uAs* action regarding the required action of an *iAs* or *dAs* action.
- When *ac* requires an iCl_1 action, and *op* already includes a sCl_pCl_1 action, the required maintenance action is also considered to be satisfied (since the specialization of an object implies that a new object of a class is created). The same reasoning has to be applied to the gCl_1Cl_p action regarding the required action of the dCl_1 action.

As it has been said before, since the UML does not provide any specific concrete syntax for defining operation specifications, we have defined our own textual notation based on the typical syntax of other action languages.

4.2.1 Determining Required Actions

This section describes how our method calculates the additional actions and/or preconditions that are necessary to ensure the maintenance of system properties for each action type. That is, given an action *a*, the goal is to characterize either the preconditions that must be checked by the operation in which the basic action will be included or the additional actions that must be executed together with *a* to ensure that *a* does not cause to evolve the system to an inconsistent state. Additional actions solve the inconsistencies that the *a* action may cause over the properties defined in the class diagram, whilst preconditions disable the execution of *a* when the inconsistencies that it may cause cannot be solved by other actions without changing the effect of the action *a*.

This process is key to ensure the strong executability of operations (wrt the properties presented in Section 3) containing potentially problematic actions. To determine how

our method should compute these additional actions or preconditions, we have carried out two steps:

1. *Identify properties to be maintained (Section 4.2.1.1).* The properties of the elements of a class diagram that may be violated when executing a certain action type are determined. For instance, if a class diagram includes a class *Cl* with a not null attribute *at*, the *iCl* action will be identified as a problematic action wrt the *isNotNull* property of *at* since the execution of *iCl* may violate this property (if the new object does not have the *at* attribute initialized).
2. *Solve or prevent inconsistencies (Section 4.2.1.2).* For each problematic <*property, action*> pair, the additional action types or preconditions that must be executed together with *action* to avoid violating *property* are determined. For instance, in this step, the *uAt* action will be signaled as a mandatory attachment to *iCl* to avoid violating the *isNotNull* property for *at* caused by the *iCl* action.

4.2.1.1 Identifying Properties to be Maintained

Some of the property values that can be defined during the specification of a class diagram (see Section 3.1) imply a constraint that must be satisfied at run-time (for instance, defining an attribute as not null implies that at run-time that attribute always has to hold a concrete value). For each one of these properties, we determine the action types that may violate them. Operations including actions of that type must be redefined to make sure their execution does not induce a violation of those constraints, as shown in the next subsection.

Let *As* be an association between classes Cl_1 (playing the role p_1) and Cl_2 (playing the role p_2) and *G* a generalization set $Gen(Cl_p, Cl_{s1}, ..., Cl_{sn})$, the list of potentially violating actions for each property are the following:

- **Minimum Multiplicity** (*min(p_2;As)*). The *dAs, uAs,* iCl_1, sCl_pCl_{si} and $uCl_{sj}Cl_{si}$ actions (such that $Cl_{si}=Cl_1$ and $Cl_{si}\neq Cl_{sj}$) may violate the constraint associated to the minimum multiplicity property when its value is greater than zero. In addition, all the iCl_x actions of any Cl_x class that inherits from Cl_1 may also violate this multiplicity.
- **Maximum Multiplicity** (*max(p_2;As)*). The *iAs* and *uAs* actions may violate the constraint defined by this property when the maximum multiplicity is lower than '*'.
- **Delete Propagation** (*delpropagation(p_2;As)*). The dCl_1, $gCl_{si}Cl_p$ and $uCl_{si}Cl_{sj}$ actions (such that $Cl_{si}=Cl_1$ and $Cl_{si}\neq Cl_{sj}$) may violate the conditions that this property establishes. Besides these actions, all the dCl_x actions of any Cl_x class that inherits from Cl_1 may also violate this condition.
- **Changeability** (*changeability(p_2;As)*). The *iAs* and *uAs* actions may violate the constraints stated by *readOnly* or *removeOnly* changeability values, and the *dAs* and *uAs* actions may violate the constraints defined by the *readOnly* and *addOnly* values.
- **isNotNull** (*isNotNull(At)*). The iCl_1, sCl_pCl_{si} and $uCl_{sj}Cl_{si}$ actions (such that $Cl_{si}=Cl_1$ and $Cl_{si}\neq Cl_{sj}$) may violate the condition that this property defines when its value is *true*. In addition, all the iCl_x actions of any Cl_x class that inherits from Cl_1 may also violate this condition.
- **isDisjoint** (*isDisjoint(g)*). The sCl_pCl_{si} action (for each subclass Cl_{si} of Cl_p) may violate the condition that this property defines when its value is *true*.
- **isCovering** (*isCovering(g)*). The $gCl_{si}Cl_p$ action (for each subclass Cl_{si} of Cl_p) may violate the condition that this property defines when its value is *true*.

4.2.1.2 Avoiding Inconsistencies

For each property constraint that may be violated by an action type *ac*, we propose either adding an **action type** that compensates the effect of *ac* to ensure the maintenance of the constraints induced by that property or a **precondition** that disables the execution of *ac* when its execution may lead to a violation of the constraint. The latter option is chosen when the former causes collateral effects that prevent maintaining the consistency of the operation while preserving the intention of the main action [11].

We present in the following tables (from Table 2 to Table 9) the maintenance actions for each action type *ac*. Each table describes, for every element of the class diagram (the *Element* column), the conditions that the element has to satisfy (*Condition* column) to be potentially affected by the execution of *ac* and the maintenance action chosen (*Required Action* column) to guarantee the fulfillment of the problematic property of the element (*Property Ensured* column) when *ac* is executed. For instance, the first row in Table 2, describes that for attributes that are not null and not derived, the *iCl(x)* action may violate its *not null* property and that to avoid this inconsistency the additional action *uAt(x,v)* must be executed after the *iCl(x)* one.

More specifically, the *Required Action* column shows either: (1) the additional actions using the notation (*direction*, *action*) where *action* is the name of the action type required, and *direction* indicates whether that action should be executed before (←) or after (→) the action analyzed; or (2) the OCL preconditions (introduced between curly brackets ({}) that must be satisfied by the system state before proceeding with the execution of the action. If several actions of the same type are needed in the first scenario, we indicate the number of times (*number* times) the action is required before the action. In addition, the parameters in the actions show the dependencies between the parameters of the analyzed action type and the parameters of its additional actions or preconditions.

Table 2 shows the maintenance actions for the *iCl(x)* basic action. When an object *x* is created in a class *Cl*, its non derived and not null attributes must be initialized after. Moreover, if the class *Cl* or its direct or indirect superclasses have a non derived association with a mandatory participation, several links (as many as the mandatory multiplicity indicates) of the association must be created after to avoid the violation of the minimum multiplicity.
As an example, in the running example, the *iConferenceEdition(x)* action of the *createConferenceEdition* operation (see Fig. 3) requires:
- the *uNumber(x,v)* action to avoid the violation of the *not null* property of the *number* attribute, and
- the *iCelebrates(x,y)* action to avoid the violation of the *minimum multiplicity* property of the *conference* role.

Note that these actions are just the actions required for avoiding the inconsistencies that the *iConferenceEdition* action can cause; the recursive process introduced at the beginning of section 4.2 must be applied to obtain the complete specification body of the *createConferenceEdition* operation. The same consideration has to be taken into account in the remainder operation examples introduced in this subsection.

Table 2. Required Actions of the *iCl(x)* action type

Element *(it has to be read: for each)*	**Condition**	**Required Action**	**Property Ensured**
$At_k \in Cl$ or $At_k \in Cl_p$ where Cl_p is a *superclass*[4] *of* Cl	*isNotNull*(At_k) and not *isDerived*(At_k)	(→, $uAt_k(x,v)$)	*isNotNull of* At_k
$As(p_1:Cl_1; p_2:Cl_2)$ where $Cl = Cl_1$ or $Cl = Cl_c$ and Cl_c is a subclass[5] of Cl_1	*min*(p_2;*As*) > 0 and not *isDerived*(*As*)	*min*(p_2;*As*) times (→, *iAs*(*x*,*y*))	*Minimum multiplicity of* p_2

Table 3 shows the maintenance actions for the *dCl(x)* basic action. The deletion of an object *x* from the class *Cl* must be forbidden if *Cl* participates in any association where the *delete propagation* property of its opposite end is marked as *restrictive* and the object *x* participates in any link of the association. This situation is avoided by the precondition defined in the first row of Table 3. The deletion of an object *x* from the class *Cl* implies the previous deletion of its links (if the opposite end of the association in which *Cl* participates is marked as *link* or *cascade*) and the deletion of the linked objects (for the *cascade* value).

The *dConferenceEdition(x)* action of the *deleteConferenceEdition* operation (see Fig. 3) requires:

- a precondition (which checks that *x* does not have any publication associated) to avoid the violation of the *delete propagation* property of the *participant* role, and
- the *dCelebrates(x,y)* action to avoid the violation of the *delete propagation* property of the *conference* role.

Table 3. Required Actions of the *dCl(x)* action type

Element *(it has to be read: for each)*	**Condition**	**Required Action**	**Property Ensured**
$As(p_1:Cl_1; p_2:Cl_2)$ where $Cl = Cl_1$ or $Cl = Cl_c$ and Cl_c is a subclass[5] of Cl_1	*delpropagation*(p_2;*As*) = restrictive	{*x*.p_2->*isEmpty*()}[6]	*Delete Propagation of* p_2
	delpropagation(p_2;*As*) = link	(←, *dAs*(*x*,*y*))	
	delpropagation(p_2;*As*) = cascade	(←, *dAs*(*x*,*y*)) (←, $dCl_2(y)$)	

Table 4 shows the maintenance actions for the *iAs (x,y)* basic action. The insertion of a link in the association *As* may require the previous insertion of one or both objects of the link for links that cannot be added after the object creations (due to the *changeability* or to the *multiplicity* of the *As* roles). Moreover, the insertion of a link must be prohibited if it violates the *maximum multiplicity* of one or both ends of the association (avoided by the precondition defined in the second row of the table).

The *iPublishes(x,y)* action of the *createPublishes* operation (see Fig. 5) requires:

[4] Superclass may be direct or indirect

[5] Subclass may be direct or indirect

[6] This precondition is slightly modified when the $dCl_1(x)$ action is performed together with the *dAs(x,y)* and $dCl_2(y)$ actions (this occurs when the *delpropagation*(p_1;*As*) = cascade). In this case the precondition does not have to take into account the link to be deleted by the *dAs* action (so, the precondition is {y.p_1.p_2->excludes(y)->isEmpty()}).

– two preconditions (which check that *x* and *y* do not have the maximum number of associated links) to prevent the violation of the *maximum multiplicity* property of the *participant* and *in* roles.

Table 4. Required Actions of the *iAs(x,y)* action type of $As(p_1:Cl_1; p_2:Cl_2)$ where $x \in Cl_1$ and $y \in Cl_2$

Element	Condition	Required Action	Property Ensured
p_1 association end (the same has to be applied for p_2)	changeability(p_1;As) = removeOnly or readOnly	(←, $iCl_2(y)$)	*Changeability of p_1*
	changeability(p_1;As) = (addOnly or unrestricted) and max(p_1;As) ≠ min(p_1;As) and max(p_1;As) ≠ *	{$y.p_1$->size() < max(p_1;As)}	*Maximum Multiplicity of p_1*
	changeability(p_1;As) = (addOnly or unrestricted) and max(p_1;As) = min(p_1;As)	(←, iCl_2(y))	

There exist certain combinations of the property values of association ends that could cause no termination of the process that expands the *iAs* action. This occurs when both association ends have minimum multiplicity values greater than 1 and either they have the same value at minimum and maximum multiplicities (row 3 of Table 4) or the *removeOnly* or *readOnly* values for the changeability property (row 1 of Table 4). In those cases the designer must specify by hand the body of the operation that contains the *iAs* action. Note that these situations are not common in real scenarios and that they can be at least detected by our method to avoid entering into an infinite loop.

Table 5 shows the maintenance actions for the *dAs (x,y)* basic action. The deletion of a link from the association *As* may require the posterior deletion of one or both objects of the link for links that cannot be removed (due to the *changeability* or to the *multiplicity* of the *As* roles). Moreover, the deletion of a link must be prohibited if it violates the *minimum multiplicity* of one or both ends of the association (avoided by the precondition defined in the second row of the table).

The *dCelebrates(x,y)* action of the *deleteCelebrates* operation (see Fig. 5) requires:
– the *dConferenceEdition(y)* action to prevent the violation of the *changeability* property of the *conference* role.

Table 5. Required Actions of the *dAs(x,y)* action type of $As(p_1:Cl_1; p_2:Cl_2)$ where $x \in Cl_1$ and $y \in Cl_2$

Element	Condition	Required Action	Property Ensured
p_1 association end (the same has to be applied for p_2)	changeability(p_1;As) = addOnly or readOnly	(→, $dCl_2(y)$)	*Changeability of p_1*
	changeability(p_1;As) = (removeOnly or unrestricted) and max(p_1;As) ≠ min(p_1;As) and min(p_1;As) > 0	{$y.p_1$->size() > min(p_1;As)}	*Minimum Multiplicity of p_1*
	changeability(p_1;As) = (removeOnly or unrestricted) and max(p_1;As) = min(p_1;As)	(→, dCl_2(y))	

Table 6 shows the maintenance actions for the *uAs (x,y,z)* basic action. The required actions are calculated from the required actions of the insertion of a link and from the deletion of a link.

The *uCelebrates(x,y,z)* action of the *updateCelebrates* operation (see Fig. 5) requires:
– the *iConferenceEdition(z)* and the *dConferenceEdition(y)* actions to prevent the violation of the *changeability* and *multiplicity* properties of the *conference* role (*changeability* is *readOnly* and the *maximum* and the *minimum* multiplicity is the same).

Table 6. Required Actions of the *uAs(x,y,z)* action type of $As(p_1{:}Cl_1; p_2{:}Cl_2)$ where $x \in Cl_1$ and $y \in Cl_2$ and $z \in Cl_2$

Element	Condition	Required Action	Property Ensured
p_1 association end	$changeability(p_1;As)$ = readOnly or $max(p_1;As) = min(p_1;As)$	$(\rightarrow, dCl_2(y))$ $(\leftarrow, iCl_2(z))$	*Changeability of p, Maximum Multiplicity of p_1, Minimum Multiplicity of p_1*
	$changeability(p_1;As)$ = removeOnly and $max(p_1;As) \neq min(p_1;As)$ and $min(p_1;As) > 0$	$(\leftarrow, iCl_2(z))$ $\{y.p_1\text{->}size() > min(p_1;As)\}$	
	$changeability(p_1;As)$ = removeOnly and $max(p_1;As) \neq min(p_1;As)$ and $min(p_1;As) = 0$	$(\leftarrow, iCl_2(z))$	
	$changeability(p_1;As)$ = addOnly and $max(p_1;As) \neq min(p_1;As)$ and $max(p_1;As) \neq *$	$(\rightarrow, dCl_2(y))$ $\{z.p_1\text{->}size() < max(p_1;As)\}$	
	$changeability(p_1;As)$ = addOnly and $max(p_1;As) \neq min(p_1;As)$ and $max(p_1;As) = *$	$(\rightarrow, dCl_2(y))$	
	$changeability(p_1;As)$ = unrestricted and $max(p_1;As) \neq min(p_1;As)$ and $min(p_1;As) > 0$	$\{y.p_1\text{->}size() > min(p_1;As)\}$	
	$changeability(p_1;As)$ = unrestricted and $max(p_1;As) \neq min(p_1;As)$ and $max(p_1;As) \neq *$	$\{z.p_1\text{->}size() < max(p_1;As)\}$	

Table 7 shows the maintenance actions for the $sCl_pCl_{si}(x)$ basic action. The specialization of an object *x* of the class Cl_p to the subclass Cl_{si} must be prohibited if *x* is an object of other subclass of Cl_{sj} of a disjoint generalization set. The object specialization requires the posterior initialization of its non derived and not null attributes, the insertion of links for non derived association with a mandatory participation and the specialization of the object to any subclass of Cl_{si}, if Cl_{si} is the superclass of a covering generalization set.

For the running example, operations including specialization actions have not been generated since the generalization set is disjoint and complete (according to Rule 4).

Table 7. Required Actions of the sCl_pCl_{si}(x) action type of $Gen(Cl_p,Cl_{s1},...,Cl_{sn})$

Element *(it has to be read: for each)*	**Condition**	**Required Action**	**Property Ensured**
$As(p_1:Cl_1; p_2:Cl_2)$ where $Cl_{si} = Cl_1$	$min(p_2;As) > 0$ and not $isDerived(As)$	$min(p_2;As)$ times $(\rightarrow, iAs(x,y))$	*Minimum multiplicity of* p_2
$At_k \in Cl_{si}$ where $Cl_{si} = Cl_1$	$isNotNull(At_k)$ and not $isDerived(At_k)$	$(\rightarrow, uAt_k(x,v))$	*isNotNull of* At_k
$g=Gen(Cl_p,Cl_{s1},...,Cl_{sn})$	$isDisjoint(g)$	$\{$not $x.IsTypeOf(Cl_{sj})\}$ $\forall$ j=1..n where $Cl_{si} \neq Cl_{sj}$	*isDisjoint of g*
$g'=Gen(Cl_{si},Cl_{si1},...,Cl_{sin})$	$isCovering(g')$	$sCl_{si}Cl_{sii'}(x)$ for any $i'=1..n$	*isCovering of g'*

Table 8 shows the maintenance actions for the $gCl_{si}Cl_p(x)$ basic action. The generalization of an object x of the subclass Cl_{si} to the class Cl_p must be prohibited if x is not an object of other subclass of Cl_p of a covering generalization set. The object generalization requires the previous deletion of its links (if the opposite end of the association in which Cl_{si} participates is marked as *link* or *cascade*) and the deletion of the linked objects (for the *cascade* value).

Again, for the running example, operations that include generalization actions have not been generated since the generalization set is disjoint and complete (according to Rule 4).

Table 8. Required Actions of the $gCl_{si}Cl_p$(x) action type of $Gen(Cl_p,Cl_{s1},...,Cl_{sn})$

Element *(it has to be read: for each)*	**Condition**	**Required Action**	**Property Ensured**
$As(p_1:Cl_1; p_2:Cl_2)$ where $Cl_{si} = Cl_1$	$delpropagation(p_2;As)$ = restrictive	$\{x.p_2->isEmpty()\}$	*Delete Propagation of* p_2
	$delpropagation(p_2;As)$ = link	$(\leftarrow, dAs(x,y))$	
	$delpropagation(p_2;As)$ = cascade	$(\leftarrow, dAs(x,y))$ $(\leftarrow, dCl_2(y))$	
$g=Gen(Cl_p,Cl_{s1},...,Cl_{sn})$	$isCovering(g)$	$\{x.IsTypeOf(Cl_{sj})\}$ where $Cl_{si} \neq Cl_{sj}$	*isCovering of g*

Table 9 shows the maintenance actions for the $u\ Cl_{si}Cl_{sj}(x,y,z)$ basic action. The required actions are calculated from the required actions of an object specialization and an object generalization.

The *uPublicationInternalDocument(x)* action of the *updatePublicationInternalDocument* operation (see Fig. 6) requires:

- a precondition (which checks that x does not have any link to a conference edition) to avoid the violation of the *delete propagation* property of the *in* role.

Table 9. Required Actions of the $uCl_{si}Cl_{sj}$ action type of $Gen(Cl_p,Cl_{s1},...,Cl_{sn})$

Element *(it has to be read: for each)*	**Condition**	**Required Action**	**Property Ensured**
$As(p_1{:}Cl_1;\ p_2{:}Cl_2)$ where $Cl_{si} = Cl_1$	$delpropagation(p_2;As)$ = restrictive	{x.p_2->isEmpty()}	*Delete Propagation of* p_2
	$delpropagation(p_2;As)$ = link	$(\leftarrow, dAs(x,y))$	
	$delpropagation(p_2;As)$ = cascade	$(\leftarrow, dAs(x,y))$ $(\leftarrow, dCl_2(y))$	
$As'(p_1{'}:Cl_1';\ p_2{'}:Cl_2')$ where $Cl_{sj} = Cl_1'$	$min(p_2';As) > 0$ and not $isDerived(As')$	$min(p_2';As)$ times $(\rightarrow, iAs'(x,y))$	*Minimum multiplicity of* p_2'
$At_k \in Cl_{sj}$	$isNotNull(At_k)$ and not $isDerived(At_k)$	$(\rightarrow, uAt_k(x,y))$	*isNotNull of* At_k
$g'=Gen(Cl_{sj},Cl_{sj1},...,Cl_{sjn})$	$isCovering(g')$	$sCl_{sj}Cl_{sjj'}(x)$ for any $j'=1..n$	*isCovering of* g'

Note that the table for the *uAt* action is not defined since the execution of this action does not cause the violation of any of the properties considered.

4.3. Specifying Operation Signatures

The last step of our method focuses on the specification of the operation signatures. Obviously, the signature depends on the actions included in the operation body. Each action may require the addition of new parameters in the signature. The basic idea is that every variable that appears as a parameter in the action must also appear as a parameter (of the same type) in the operation so that a designer can provide its value. Four exceptions apply:

1. Object variables for the *iCl* action are not parameters of the operation. These new objects are created *during* the operation execution.
2. A parameter variable that has already appeared in a previous action does not generate a new operation parameter (i.e., if an operation consists of two events, $iAs(x_1,x_2)$ and $iAs(x_1,x_3)$, only three parameters x_1, x_2 and x_3 are defined).
3. We use the implicit parameter *self* as a replacement for one of the parameters whose type is the class to which the operation is attached (i.e., if an operation defined in a class *Cl* has the event $uAt_i(x,v)$, only a parameter for *v* is generated; the implicit *self* parameter is used whenever *x* appears).
4. Variables for actions that can be obtained by *self* are not parameters of the operations. For example, variables for *dAs* actions included in a *deleteCl* operation are not parameters of the operation. In those cases, the link/s to be deleted are the ones in which the *self* parameter participates, and thus they can be determined automatically.

4.4. Application to the Running Example

In this subsection we apply our method to the example of Fig. 2. In Fig. 3, 4, 5 and 6 we have already shown the list of operations generated for the example. Now in what follows we introduce the complete specification for each operation. Comments to clarify the maintenance actions required for the operation are added if necessary. We present the operations grouped by the rule that generates them.

Operations generated by the application of Rule 1 (Classes).

- For the *Publication* class, the operations generated are *createPublication* and *deletePublication*:

```
Publication::createPublication(vname:String){
  iPublication (p); --main action
  uName(p,vname); --avoiding not null constraint violation}
```

The *createPublication* operation specification includes two basic actions. The first one, *iPublication,* corresponds to the basic action of a creation operation (according to Table 2). The second one, *uName*, is defined to avoid the violation of the *isNotNull* property of the *name* attribute (see row 1 of Table 2). This update action does not require further maintenance actions.

```
Publication::deletePublication(){
  if (self.in->isEmpty()) then
       dPublication(self); --main action
  endif; }
```

The *deletePublication* operation specification includes a precondition that prevents the deletion of a publication when it has a conference edition associated to it (see row 1 of Table 3).

- For the *ConferenceEdition* class, the operations generated are *createConferenceEdition* and *deleteConferenceEdition*:

```
ConferenceEdition::createConferenceEdition(vnumber:String,
conf:Conference){
  iConferenceEdition (e); --main action
  uNumber(e,vnumber); --avoiding not null constraint violation
  iCelebrates(e,conf); --due to the min multiplicity }
```

uNumber is defined to avoid the violation of the *isNotNull* property of the *number* attribute. *iCelebrates* is added to satisfy the minimum multiplicity value of the *conference* role (see row 2 of Table 2).

```
ConferenceEdition::deleteConferenceEdition(){
  if (self.participant->isEmpty()) then
       dCelebrates(self,self.conference);--delete propagation
       dConferenceEdition(self); --main action
  endif; }
```

The *deleteConferenceEdition* operation specification includes a precondition that prevents the deletion of a conference edition when it has a publication associated to it. *dCelebrates* is included to delete the link between the conference edition to be deleted and its conference (see row 2 of Table 3).

- For the *Conference* class, the operations generated are *createConference* and *deleteConference*:

```
Conference::createConference(vname:String){
  iConference (c);--main action
  uName(c,vname);--avoiding not null constraint violation }
```

uName, is defined to avoid the violation of the *isNotNull* property of the *name* attribute. This action does not require any other action.

```
Conference::deleteConference(){
  if (self.edition.participant->isEmpty()) then
       foreach ConferenceEdition e in self.edition do
            dCelebrates(e, self);--delete propagation cascade
            dConferenceEdition(e);--delete propagation cascade
       end for;
       dConference(self);--main action
  endif; }
```

The *deleteConference* operation specification includes an iterative statement that deletes all links to conference editions of a conference as well as deletes all the associated conference editions. This is because the delete propagation value of the *edition* role is *cascade* (see row 3 of Table 3). Moreover, the operation includes a precondition that prevents the deletion of conference when it has conference editions associated that in turn have publications associated to them. This is because of the value *restrictive* at the delete propagation of the *participant* role (see row 1 of Table 3). The last action of the operation is the one that is related to the delete operation, *dConference*.

- For the *InternalDocument* class, the operations generated are *createInternalDocument* and *deleteInternalDocument*:

```
InternalDocument::createInternalDocument(v:String){
  iInternalDocument(i);--main action
  uName(i,v);--avoiding not null constraint violation }
```

uName, is defined to avoid the violation of the *isNotNull* property of the *name* attribute. This action does not require any other action.

```
InternalDocument::deleteInternalDocument(){
  dInternalDocument(self);--main action }
```

Operations generated by the application of Rule 2 (Attributes).

```
ConferenceEdition::updateYear(v:Integer){
   uYear(self,v); --main action }

ConferenceEdition::updateNumberOfSubmissions(v:Integer){
   uNumberOfSubmissions(self,v); --main action }

Publication::updatePublicationDate(v:String){
   uPublicationDate(self,v); --main action  }

Document::updateDescription(v:String){
   uDescription(self,v); --main action  }
```

Operations generated by the application of Rule 3 (Associations).

- For the *Publishes* association, the operations generated are a *createPublishes* operation in the *ConferenceEdition* and *Publication* classes:

```
Publication::createPublishes(edt:ConferenceEdition){
  if (edt.participant < 100) and (self.in < 1) then
            iPublishes(self, edt);--main action
  endif }

ConferenceEdition::createPublishes(pub:Publication){
  if (pub.in < 1) and (self.participant < 100) then
            iPublishes(pub, self);--main action
  endif }
```

  The *createPublishes* operation specification includes a precondition that prevents the violation of the maximum multiplicity property of the *participant* and *in* roles (see row 2 of Table 4).

- For the *Celebrates* association, a *createCelebrates* operation, a *deleteCelebrates* operation and an *updateCelebrates* operation are generated in the *Conference* class:

```
Conference::createCelebrates(vNumber:String){
  iConferenceEdition(e);--due to the max multiplicity
  uNumber(e,vNumber);--avoiding not null constraint violation
  iCelebrates(e, self); --main action }
```

  The *createCelebrates* operation specification includes the creation of a new conference edition. This is because of the value *readOnly* of the *changeability* property at the *conference* role (see row 3 of Table 4), which means that conference editions cannot add celebrate links throughout their live (so, the creation of a *celebrate* link is just possible together with the creation of a conference edition). The operation also includes the *uName* action to avoid the violation of the *isNotNull* property of the *name* attribute.

```
Conference::deleteCelebrates(edt:ConferenceEdition){
  if (edt.participant->isEmpty()) then
        dCelebrates(edt, self);--main action
        dConferenceEdition(edt);-- due to readOnly changeability
  endif; }
```

  The *deleteCelebrates* operation specification includes the *dConferenceEdition* action to prevent the violation of the *changeability* property at the *conference* role (see row 1 of Table 5). Note that since a *celebrates* link cannot be removed from a conference edition (changeability(*conference, celebrates)* = *readOnly*), in order to delete a *celebrates* link it is necessary that the *deleteCelebrates* operation also deletes the conference edition involved in that link. In addition, since the *dConferenceEdition* action can violate the *delete propagation* of the *participant* role, a precondition to avoid this violation has been included.

```
Conference::updateCelebrates(edt:ConferenceEdition,
vnumber:String){
```

```
  if (edt.participant->isEmpty()) then
       iConferenceEdition (e);
       uNumber(e,vnumber);
       uCelebrates(self, edt, e);
       dConferenceEdition(edt);
  endif;
}
```

The *updateCelebrates* operation specification includes the *iConferenceEdition* and the *dConferenceEdition* actions for creating a new conference edition to be linked to the conference and deleting the conference edition `edt` linked to the conference. This allows preventing the violation of the *changeability* and *multiplicity* properties at the *conference* role (see row 1 of Table 6). The *iConferenceEdition* action requires in turn the *uNumber* action to prevent the violation of the *isNotNull* property of the *number* attribute of the *ConferenceEdition* class. Also, the *dConferenceEdition* action requires a precondition to prevent the deletion of a conference edition when it has a publication associated to it. Note that although the *iConferenceEdition* action requires an *iCelebrates* action and the *dConferenceEdition* action requires a *dCelebrates* action these actions are not included in the *updateCelebrates* operation since these dependencies are satisfied by the *uCelebrates* action.

Operations generated by the application of Rule 4 (Generalizations).

```
InternalDocument::updateInternalDocumentPublication(){
   uInternalDocumentPublication(self); --main action}

Publication::updatePublicationInternalDocument(){
   if (self.in->isEmpty()) then
        uPublicationInternalDocument(self);--main action
   endif }
```

The *updatePublicationInternalDocument* operation specification includes a precondition that prevents the modification of the specialization of a publication when it has a conference edition associated to it (see row 1 of Table 9).

5. Formalizing the method as a model-to-model transformation

Our method has been defined as a M2M endogenous transformation that takes a specific UML class diagram as input and outputs the same class diagram extended with the set of strongly executable operations that suffice to provide a basic behavior for the system under development. The source model (and the target one, since it is an endogenous transformation) is an instance of the UML metamodel possibly annotated with our profile. Note that the target model is still a Platform Independent Model (according to the MDA terminology) since it is completely platform independent. The refinement introduced by the transformation does not add any technology-specific details.

The formalization of the M2M transformation is defined in ATL [9]. ATL is a hybrid model transformation language developed by the ATLANMOD Group [43]. Our set of

ATL transformation rules automatically transform the elements of a source class diagram into elements in the target class diagram according to the steps described in the previous section. ATL provides a compiler and a virtual machine that enables the execution of ATL transformations.

ATL provides the *ATL modules* to implement transformations. An ATL module specifies the rules that define how the elements of the input model are mapped to elements of the output model. A declarative rule in ATL specifies for an input element (defined in the "from" section of the rule) how the output elements should be generated (defined in the "to" section of the rule). Moreover, ATL modules allow metamodel extensions to be specified in order to define computed attributes (*attributes*) or operations (*helpers*).

5.1. Structure of the Transformation

The complete description of the ATL transformation can be found in [44]. In this section we just summarized some of its main elements.

As a first step, we have structured our transformation in an ATL module that defines a new model[7]. The header of the transformation within the eclipse-based ATL IDE is the following:

```
module GeneratingOperations;
create output : OUT from input : IN;
```

Although input (IN) and output (OUT) metamodels can be seen in the transformation specification, both of them are in fact associated to the same metamodel when the transformation is launched. The metamodel used to express the transformation is the UML metamodel slightly extended to allow the definition of the enriched association properties we have been using throughout the paper.

Then, for each type of element that can be contained in the input model (*Classes*, *Associations*, *Attributes*, *GeneralizationSets*), we define a set of ATL rules to generate the corresponding elements in the output model following the rules introduced in section 4.1.

As an example, we show the ATL rule for generating *creation* and *deletion* operations for classes explained in Section 4.1:

```
rule ClassNotAbstract2ClassCreateOpDeleteOp{

from
	c: IN!Class (not c.isAbstract and not c.isDerived and not
	c.is_SuperclassCovering and not c.is_InRestrictiveAs)
to
	oout : OUT!Class (
		name <- c.name,
		isAbstract <- c.isAbstract,
		isDerived <- c.isDerived,
		associationEnd <- c.associationEnd,
```

[7] This implementation creates a new output model that clones the input model and extends it with the new operations. Nevertheless, the transformation can also be implemented extending the input model (without creating a new model).

```
            system <- c.system,
            operation <- Sequence{createOp,deleteOp}
        ),
        createOp: OUT!Operation (
            name <- 'create' + c.name,
            body <- c.build_CreationBody(thisModule.initialize_actionList),
            parameter <-
            c.build_CreationParameters(thisModule.initialize_actionList),
            system <- c.system
        ),
        deleteOp: OUT!Operation (
            name <- 'delete' + c.name,
            body <- c.build_DestroyBody(thisModule.initialize_actionList),
            parameter <-
            c.build_DestroyParameters(thisModule.initialize_actionList),
            system <- c.system
        )
}
```

First, the static elements of the source class are cloned in the return class object (since the static aspects are not modified by the transformation). Secondly, create and delete operations are added if necessary (note that the *from* clause in the rule restricts the application of the rule to classes that are not abstract, not superclass of a covering generalization, and not participating in restrictive or mandatory associations, according to Rule 1).

The auxiliary helper operation *build_CreationBody* is in charge of providing the imperative description of the creation operation's behavior. As an example, we show the implementation of the *build_CreationBody* helper:

```
helper context IN!Class def : build_CreationBody(actionList :
Sequence(String)) : String =
        let _notNullAttributes: Sequence (IN!Attribute) = self.attribute->
            select(a |a.isNotNull) ->union(self.get_InheritedAtNotNull)
        in
        self.text_iCl +
        if _notNullAttributes->flatten()->isEmpty() then
            if self.get_MandatoryClasses(actionList->append(self.text_iCl))->
            notEmpty() then
                    self.get_MandatoryClasses(actionList->
                    append(self.text_iCl))->
                    collect(y|y.build_CreationLinkBody(actionList->
                    append(self.text_iCl)))->sum()
            else
                    ' '
            endif
        else
            _notNullAttributes->flatten()->collect(a| a.text_uAt)->sum() +
            if self.get_MandatoryClasses(actionList->append(self.text_iCl))->
            notEmpty() then
                    self.get_MandatoryClasses(actionList->
                    append(self.text_iCl))->
                    collect(y|y.build_CreationLinkBody(actionList->
                    append(self.text_iCl)))->sum()
            else
                    ' '
            endif
        endif;
```

Note how depending on the properties of the class and the associations, the body will include more or less required actions for the creation operation.

More specifically, the helper builds a string that contains the textual behavior specification of the creation operation. This string is created by concatenating:

- (1) the textual specification of the main action of the operation (*iCl()*).
- (2) the textual specification of the actions in charge of fulfilling the not null property of the attributes of the class (*uAt()*).
- (3) the textual specification of the actions in charge of fulfilling the mandatory participation of the class in associa*tions (iAs()).* This specification is obtained using the *build_CreationLinkBody* helper.

Similar rules and helpers have been defined for generating the remainder of the operations of the class diagram of the output model [44].

5.2 Transformation Validation

We have validated the transformation by checking that it behaves as intended, which is a common kind of validation [45]. One way to do so is to apply the transformation to a set of model examples and to check the transformation does what it was intended to do (i.e. to compare the expected results with the generated ones).

Following this approach, we have validated our transformation by means of testing its behavior using as input several case studies used in the Software Engineering course of the School of Applied Computer Science of the Universidad Politécnica de Valencia. The description of the case studies can be found in [44]. The results returned by the transformation have been compared with the expected ones defined by ourselves according to the method developed in [46].

5.3 Tool Implementation

The previous transformation has been implemented as an Eclipse plug-in within the open-source modeling framework *MOSKitt*. The *MOSKitt* project with the transformation can be downloaded from [44].

As an example, Fig. 7 and 8 show the execution of the implemented transformation over our running example. As shown in Fig. 7 the output model contains the same static elements as the input model but adds 19 new operations (the generated operations are those indicated in Fig. 3, 4, 5 and 6). The transformation has also generated a string for each operation that contains the description of its behavior in the property *body*. Fig. 8 shows the value of the *body* and *parameter* property for the *creation* operation of the *ConferenceEdition* class. The body of the operation is generated by means of the helper *build_CreationBody* shown in section 5.1.

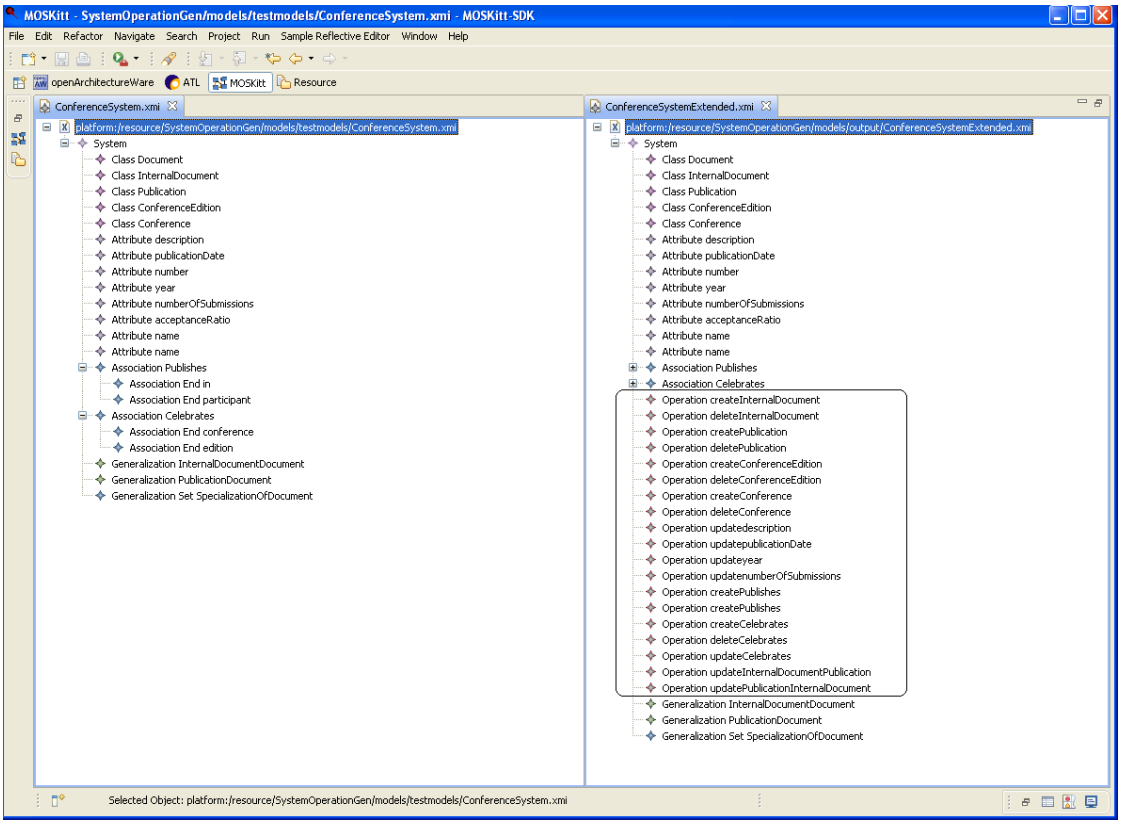

Figure 7. Representation of the Input and Output models of the case study in the model tree editor

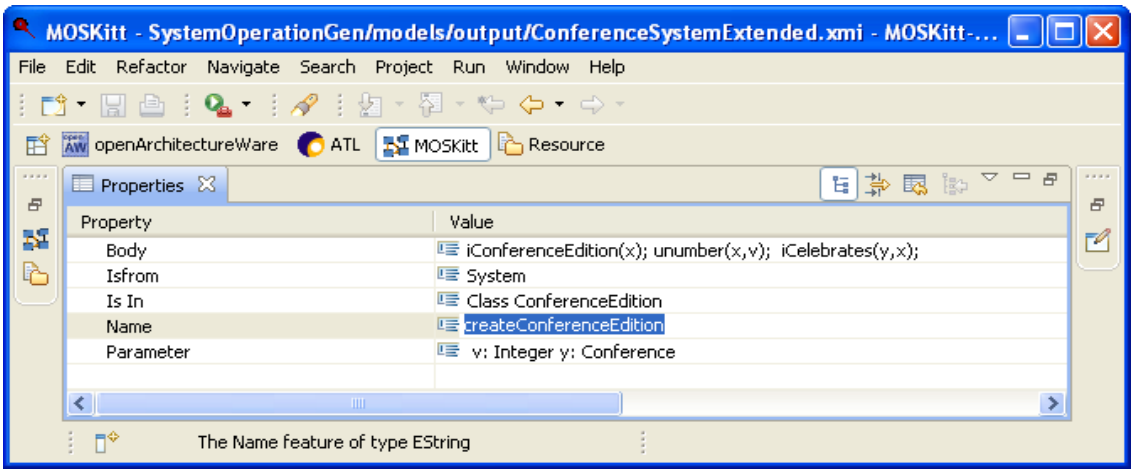

Figure 8. Description of the body and parameters of the *createConferenceEdition* operation

6. Method Evaluation

We have applied our method to different scenarios in order to evaluate its usefulness in terms of the completeness and quality gain of our automatically generated specification when comparing it with a manual specification and with the results provided by generation code tools for the same class diagram.

The first analysis evaluates the completeness of our method (i.e. the percentage and richness of operations that our method is able to generate in comparison with the ones that: 1 - a designer would manually define and 2 - generated by other code generation tools). A second analysis focuses on the quality improvement that can be expected when

using our method by comparing the operations and the constraints detected and handled by each operation when generated by our method with those detected and correctly handled by students when manually implementing the same operations.

As can be seen in the next subsections, results are satisfactory in both situations. Our method suffices to automatically generate a large number of operations in all analyzed situations and is very useful to avoid mistakes during the definition of the operations.

6.1. Evaluation of the Method Completeness

This first analysis consisted in comparing the set of operations originally specified by the designers of a real-life application with the set of operations generated automatically for the same application. This comparison is based on the well-known *osCommerce*[47] online shop e-commerce solution used by more than 200000 online stores.

The main objective of this analysis was to evaluate the percentage of operations that are completely covered by our method with respect to the total number operations designed during the original system specification. In the comparison, we also include the results obtained when using two alternative popular MDD tools, IBM Rational Architect [26] and Poseidon [22], featuring behavioral generation capabilities as well.

The *osCommerce* is an e-commerce solution available as free software under the GNU General Public License. The *osCommerce* project was started in March 2000 in Germany and since then, it has become the base of thousands of online stores around the world. For comparison purposes, we have analyzed a fragment of the class diagram of *osCommerce* [44] consisting of 12 classes, 7 associations, 1 generalization set and 43 attributes. For this class diagram, our method has generated 89 operations (see [44]). This accounts for a 85% of the total number of operations originally specified by the designers of the system. From these 85%, in a 84% the specification generated by our method is completely equivalent to the one in the original system. For the other 1% we get a partial definition (the original condition included some degree of additional business logic that could not be automatically derived from the class diagram information). Clearly, this shows that with our method designers could have avoided the generation of most of the system operations resulting in a significant productivity improvement during the development.

In comparison, results obtained when using IBM Rational Architect and Poseidon tools are much worse. These tools generate only *getter* and *setter* methods for each attribute and association end of the UML class diagram (Java, C#, C++, and other programming languages can be chosen to implement these methods). In the case of IBM Rational Architect, the *setter* methods just set new values (objects) to the attributes (association ends) without checking any possible constraint violation. In Poseidon tool, *setter* methods only guarantee the uniqueness of the attribute values (association end objects). Comparing these results with the operations generated by our method, we see that our approach generates all those setter operations plus 54 additional modification operations (to create and destroy objects, for instance) and, most importantly, our operations take care of the system integrity by checking or maintaining all constraint violations during the operation execution which is not the case for the IBM Rational Architect or Poseidon tools.

We have confirmed these completeness results by repeating the study with two more case studies, *CMA* [48] and *EmpTraining* [44]. In both cases, the results have been similar: a 69% of the operations specified for *EmpTraining* and *CMA* are completely generated by our method and a high percentage of the remaining ones (21% for *EmpTraining* and 31% for *CMA*) are partially generated as well. We refer the reader to [7] for a more detailed description of these experiments.

6.2. Evaluating the Quality gain when using our Method

The second analysis studies several applications developed by undergraduate students during their last year in an Applied Computer Science degree. The applications were developed in the Visual Studio .Net 2005 environment and implemented using the C# language. The input for developing the systems was a set of UML class diagrams describing the systems to be implemented (see [44]).

For each student application, we compare the behavior of the application with the behavior generated by our method to analyze: (1) the percentage of life cycle operations that were missing in the system designed by the student and (2) the percentage of the model property constraints that are not guaranteed to be fulfilled by the student's systems. The goal was to demonstrate that our method can be applied to detect and avoid many of the common errors made during the implementation of system operations when taking only the static part of UML models as a source.

We have analyzed 10 applications for two case studies (CS1 and CS2). These applications have been randomly selected. Each application has been developed in groups composed of one or two students. Fig. 9 and Fig. 10 summarize the results of the evaluation for CS1 and CS2 respectively.

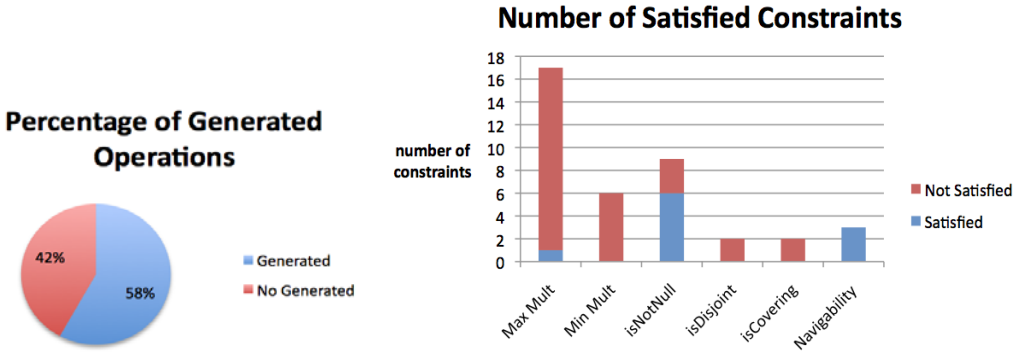

Figure. 9. Results of the analysis for CS1

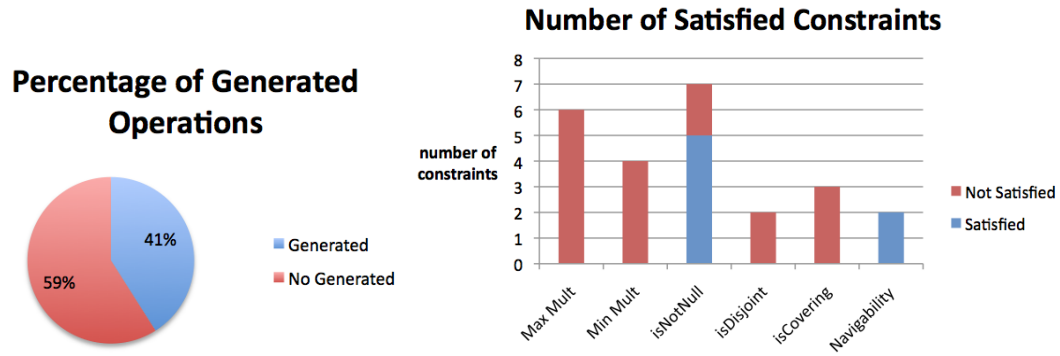

Figure. 10. Results of the analysis for CS2

The results of the analysis reveal that:

- an important number of operations are missing in the student applications (42% for CS1 and 59% for CS2 of the necessary operations to perform all typical create/delete/update changes was not implemented), and
- a high number of constraints are neither guaranteed to be fulfilled: for example just one maximum multiplicity constraint is guarantee in CS1, none in CS2 and the only property that is fully guaranteed to be satisfied at both CS1 and CS2 is the navigability property.

Note that the delete propagation and changeability properties are not included in the study (see graphics on the right of Fig. 9 and 10). The former is not included since the class diagrams used by the students were UML class diagrams where delete propagation was not specified. The last property is not included since its value at the class diagrams of the case studies was always *unrestricted* and, therefore, its fulfillment was always guaranteed. Moreover, the study points out several interesting issues. On the one hand, navigability is always guaranteed to be fulfilled (students correctly avoid implementing operations to manage associations in classes with a non-navigable association end). The reason for the high-success, in this particular case, could be that non-navigable ends always appear as part of aggregation associations and this kind of associations got a special emphasis during the course. On the other hand, *maximum* and *minimum multiplicity constraints*, and *isDisjoint* and *isCovering* constraints were almost completely ignored. We believe that this shows that students did a very limited set of tests with the application. Otherwise they should have detected these violations. It is also important to note that no student implemented operations not identified and specified by our method, which implies that for these scenarios our method was complete.

We believe these empirical evaluations clearly show the benefits of our method. If students had been given, instead of a simple static UML diagram, a class diagram extended with our operations they would have produced a software system with a considerable better quality, since this additional information would have help students to significantly reduce the number of errors made during the implementation phase and would have considerably reduced the time spent in doing it. Note that expert programmers would have better performed the application development, i.e. would have specified a higher number of operations and guaranteed a higher number of constraints. Nevertheless, our method does generate valuable operations which are always correct. Thus, the tool is useful for expert programmers, too. For instance, even if maybe they would not have made mistakes when defining the operations we can save them time by generating them automatically

7. Related Work

The (semi)-automatic derivation of system behavior from different aspects of a system has been faced from different perspectives. A summary table at the end highlights the most relevant aspects of this comparison.

In the deductive database field, specifically in approaches that cope with integrity checking or integrity maintenance problems at compile-time, we find approaches aimed at extending transactions/operations with preconditions or additional actions to always

ensure their successful execution. Among others, in [11] and [12], the authors adapt the state based approach to formal specifications supporting explicitly the concepts of state and state transition. In those cases, the expressivity of the logical language used for the definition of the structural schema and the operations is poorer than what can be expressed in UML. Concretely, they do not deal with properties related to associations and attributes (as for instance, read only properties of attributes and association navigability). In those works, inconsistent transactions/operations provided by designers are replaced, if possible, by new consistent ones preserving the intended effect of the old ones. Therefore, these approaches require the designer to provide a first version of the operations instead of generating them from scratch and do not enforce the attribute and association properties, as our method does. Moreover, none of the previous approaches has been implemented. In the same field, [13] uses a declarative logic-based approach (implemented in Prolog) to define the structural part of a system and the transactions that can be executed on it. As before, the expressiveness of the structural schema is limited and the generated operations only consist of insert and delete actions (update ones are not supported) that ignore most of the integrity constraints except for minimum multiplicities and derived elements.

In the active database field, several approaches suggest the use of triggers to ensure the consistency enforcement of transactions (on the contrary, our method does not need triggers for guarantying the fulfillment of the properties of the class diagram). Remarkable works in this field are [14] and [15]. In [14], the authors adopt Domain Relational Calculus as the underlying language in the expression of constraints and transaction actions and Relational Algebra for the database definition. The method generates automatically a set of actives rules that may be used to enforce the set of defined constraints. Thus, when a transaction is executed the method determines a partial order on the active rule set to guarantee the termination in a final state such that all violated constraints are corrected. If the partial order cannot be determined then the transaction is aborted. In this approach transactions must be provided by designers and association and attribute properties are not covered except for minimum multiplicities and derivability property. The method proposed in [15] extends the previous one. It uses SQL based syntax to express constraints. This syntax permits to define more powerful constraints as maximum multiplicities. Instead, the method only generates the event and the precondition part of the set of active rules that may be used to enforce the set of defined constraints. The action part of rules (i.e. the actual behavior of the operation) has to be added manually by the designer.

In the conceptual modeling field, some proposed techniques aim to generate operations from structure diagrams but differ from our method in several aspects. The study in [16] partially determines the set of possible basic actions to be applied to a UML class diagram (generalizations and changeability and delete propagation properties are not considered). For these actions, only preconditions (and not additional actions) are generated when they are necessary. In this approach, the modeling language used is the B formal notation [49]. In [17], a set of basic operations (similar to our concept of basic actions) and a set of elementary operations (similar to our concept of operation) composed of basic operations are derived from an EER diagram. The enforcement of these operations is achieved by means of update propagations. These operations are not necessarily executable since cardinality constraints and other explicit constraints are not considered in any case and, thus, preconditions to guarantee these constraints have to be added by hand. The work presented in [18] generates the declarative definition of a set

of structural events to be applied to a structural model with dynamic features expressed in the ROSES language. Although this approach deals with association properties equivalent to ours, generalizations (with the disjoint and completeness constraints) and navigability are not considered. In [19], an approach to automate the extension of the declarative operation specifications taking into account only association invariants is presented. The authors use the *Booster* notation to define object models and the specifications of the operations. Unlike our proposal, this work just deals with referential integrity, multiplicity and symmetry properties.

Additional approaches, as [20] and [21], extend the operations with a set of preconditions to ensure the executability of the operations. In this case, operations are not generated automatically, as our method does, but the designer is responsible for providing the set of operations. In [20] the preconditions are extracted from the set of constraints defined in the structural model (a subset of the OMT object model with only classes and associations is considered). This approach only works with properties and constraints that may be expressed in Z. Generalizations and changeability and delete propagation properties are not considered. In [21] the preconditions are extracted from a set of predefined constraints. In this work, minimum and maximum multiplicities of associations and disjoint and complete constraints are covered but not the others that our method considers. Alternatively, other approaches try to generate system operations from the information provided in different diagrams, such as the use case diagram. For instance, [50] presents a method for generating system operations from use cases specifications. Nevertheless, this method is not automatic and the specification of each system operation must be provided manually.

A more technological approach related to our work is the one of Akehurst et al. in [51], which provides a set of patterns that allow generating automatically Java code from UML class diagrams. The patterns cover the implementation of UML 2.0 associations generating methods to manage them at run-time. These methods include assertions to check that the constraints specified in the UML 2.0 associations are satisfied, whereas our proposal goes by covering not only associations but the rest of model elements in UML and by adding the possibility of having an integrity maintenance solution that compensates the effect of the method behaviour instead of just checking the system state at the end of the operation. Moreover, this work targets a specific programming language, which hampers its reuse in other technologies. On the contrary, our work generates a UML model that it is still at the PIM level, i.e. completely platform independent. The operations introduced by our method do not add any technology-specific details.

Finally, we have analyzed most MDD tools (commercial or open source) able to generate methods following a similar philosophy as the one tackled in this paper. Nevertheless, the number and content of operations generated by these tools is lesser than what it is achieved by our method. Tools such as Poseidon [22], Enterprise Architect [23], Fujaba [24], Modelio [25] or IBM Rational Architect [26] provide MDD solutions that allow starting the software development process by specifying a class diagram of the system under construction. Then, they provide model to text transformations that generate code from that diagram including methods for the classes in the specified classes. However, these methods are just “basic” *getter* and *setter* methods for managing attributes and associations (other kinds of methods, e.g. to manage generalizations, are not supported). By “basic” we mean methods that these

methods just include in their body the actions to perform the functionality of the method but do not include the necessary checking conditions or additional functionality to guarantee that the properties and constraints specified in the UML class diagram are satisfied. As a rare exception, Modelio [25] tool provides a library that allows generating code that (only) guarantees that cardinality constraints are satisfied. Therefore, when using these tools, the designers/programmers have to manually make the generated operations executable. We believe these tools could implement the method presented in this paper to improve their generation process. Also, note that current MDD tools target always a specific programming languages and/or technology platforms and thus, their results are hardly reusable for other technologies.

A specially relevant tool is the OO-Method approach [27], an OO MDD method in whose development some of the authors have participated. OO-Method is supported by the ONME commercial tool [52], which generates system operations to manipulate elements specified in a conceptual schema. However, this conceptual schema is defined using a proprietary language and the implementation of the operations only guarantees a subset of the properties/constraints that are handled in this paper. Moreover the definition of the operations is not performed at the conceptual level but using concrete implementation languages to generate code which makes difficult the reusability of their method. We plan to improve OO-Method with the techniques developed in this work.

Table 10 summarizes the most relevant aspects of the methods and tools previously reviewed. For each of them we have identified the inputs of the method (structural part plus input behavior model if the method does not generate the operations from scratch) and its output (what parts of the behavioural aspects are generated and the constraints and properties considered to generate them). We also indicate if there is a tool implementing the theoretical concepts presented in the method.

Table 10. Comparative table of related work

	Method Input		**Method Output**		**Tool available**
Methods and Tools	**Structural part (modeling language)**	**Behavioral part (modeling language)**	**Behavioral part**	**Constraints and Properties Considered**	
[11]	Logical language	User defined basic actions (logical language)	Required actions	Uniqueness, inclusion/exclusion, object-generating	×
[12]	Logical language	User defined basic actions (logical language)	Required actions	Uniqueness, inclusion/exclusion, object-generating	×
[13]	Logical based language	User defined basic actions except for updates (logical language)	Required actions and preconditions	Constraints expressed as closed first-order formulas (recursive rules and aggregate functions not covered)	√
[14]	Relational algebra	User defined transactions (DB update language)	Active rules - event, preconditions and required actions	Constraints expressed as relational calculus formulas (recursive rules and aggregate functions not covered)	√
[15]	Relational algebra	User defined transactions (DB update language)	Active rules - event and preconditions	Constraints expressed in a declarative language similar to SQL (recursive rules and aggregate functions covered)	×
[16]	UML without generalizations	-	Basic actions and their preconditions	Constraints expressed in B language	√
[17]	EER	-	Basic and elementary operations with required actions	EER constructs (minimum and maximum mult. and some general constraints not covered)	×
[18]	ROSES object model	-	Structural events with preconditions	Constraints expressed in ROSES clauses (minimum and maximum mult., derivability, common	×

				dynamic constraints of objects and attributes)	
[19]	Booster language	-	Operations with preconditions and postconditions	Graphic constraints (referential, multiplicities (only 0..1,1,*) and symmetric constraints)	×
[20]	Subset of OMT (only classes and relations)	Basic actions (Z language)	Basic action preconditions	Constraints expressed in Z language (disjoint and complete constraints and changeability and delete propagation properties not covered)	×
[21]	UML	Basic actions (OCL)	Basic action preconditions	Identifier, irreflexive, symmetric, asymmetric, antisymmetric, acyclic, path inclusion, exclusion and equality, value comparison, disjoint and complete and minimum and maximum mult.	×
[50]	UML	Use cases	System operations (their specification is not provided)	-	×
[22]	UML	-	Association end getter/setter	Uniqueness property and association navigability	√
[23]	UML	-	Association end getter/setter	Association navigability and mult. (but no bounds check)	√
[24]	UML	-	Association end getter/setter	Association navigability and mult. (but no bounds check)	√
[25]	UML	-	Association end getter/setter	Association navigability and mult.	√
[26]	UML	-	Association end getter/setter	Association navigability	√
[27]	Proprietary Language	-	Create/Destroy for classes, Create / Update/ Delete for Association Ends	Abstract classes, mult., derived roles and attributes, delete propagation, changeability and navigability	√
Our method	Enriched UML	-	Basic actions and operations with preconditions and required actions	ReadOnly, derivabiltiy and isNotNull attribute properties, abstract class property, minimum and maximum mult., derivability, navigability, changeability, delete propagation association end properties and disjoint and complete constraints	√

8. Conclusions

In this work we have defined a method (formalized as a M2M transformation using ATL) that generates a set of basic operations for an initial purely static conceptual schema. The operations generated by our method suffice to cover all basic modification operations (insert/update/deletes…) for the system under development. The number and behavior of the operations is deduced from the characteristics of the structural elements (classes, associations and so forth) in the input schema. Our construction process guarantees that no irrelevant operations are defined and that all created operations are strongly executable with respect to the most common structural properties in CSs (as multiplicity property). Our operations can also be used as a foundation to build more complex operations.

Our proposal improves the quality and productivity of the behavior specification task of the software development process by automating most of it and ensuring the absence of errors that could be introduced by designers in a manual specification. Furthermore, our method can also be applied to domain models that are derived from ontologies using the proposal presented in [53], where a framework that integrates UML class-based models and OWL ontologies is introduced. Also our approach can enhance other techniques that aim at automating model transformations. For instance, our method can be used for enhancing the automation of domain modeling in [54].

With regard to future work, we consider extending our method to deal with additional properties and constraints that are not studied in this work, for example, those regarding association classes. We also plan to study the integration in our generation process of other information sources to provide more richer and complex operations (as the use case diagrams). As an example, we will consider the approach presented in [20] that uses use case diagrams to generate parts of the behavior specification. Additionally, the ideas introduced in this paper could be applied to the verification of existing behavior schemas by means of the analysis of their completeness and executability properties as done in [55] or to detect inconsistencies between elements defined in the conceptual schema. Finally, we would like to improve the maintenance phase of the system development as well by analyzing how to incrementally regenerate the behavior schema after evolutions on the structural schema.

Acknowledgments

The authors want to thank the anonymous referees of this journal for their interesting suggestions. This work has been partly supported by the MICINN under projects TIN2008-00444, Grupo Consolidado and TIN2010-18011, and by the Generalitat Valenciana under the project ORCA PROMETEO/2009/015, and co-financed with European Regional Development Fund.

References

[1] Object Management Group, UML 2.2 Superstructure Specification, OMG Adopted Specification, (2009).

[2] Object Management Group, Object Constraint Language (OCL) 2.0, OMG Adopted Specification (2006).

[3] P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. In: *ACM Transactions on Database Systems 1(1), pp. 9-36*, (1976).

[4] T. Halpin. Information Modeling and Relational Databases. From Conceptual Analysis to Logical Design. *Morgan Kaufmann Publishers,* (2001).

[5] D. Spinellis. UML Everywhere. In: *IEEE Software 27 (5), pp. 90-91,* September (2010).

[6] B. Dobing, and J. Parsons, How UML is Used, In: *Communications of the ACM* 49(5), May (2006).

[7] M. Albert, C. Gómez, J. Cabot, V. Pelechano, Automatic generation of basic behavior schemas from UML class diagrams, In: *Software and Systems Modeling 9(1*), *pp. 47-67*, (2010).

[8] M. Albert, V. Pelechano, J. Fons, M. Ruiz, and O. Pastor, Implementing UML Association, Aggregation and Composition. A Particular Interpretation based on a Multidimensional Framework, In: *Proc. of the Advanced Information Systems Engineering (CAiSE'03),* (2003).

[9] J. Bézivin, F. Jouault, and D. Touzet, An Introduction to the ATLAS Model Management Architecture, *Research Report LINA,* (2005).

[10] MOSKitt, MOdeling Software Kitt, 14 Jan 2011, <www.moskitt.org>

[11] K. Schewe, and B. Thalheim, Towards a theory of consistency enforcement, In: *Acta Informatica 36, pp. 97-141*, (1999).

[12]S. Link, Consistency Enforcement in Databases, In: *Semantics in Databases, pp. 201-213,* (2003).

[13] J. A. Pastor, and A. Olivé, Supporting transaction design in conceptual modelling of information systems, In: *Proc. of Advanced Information Systems Engineering, pp. 40-53*, (1995).

[14] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic Generation of Production Rules for Integrity Maintenance, In: *ACM Transactions on Database Systems, 19(3), pp. 367-422,* (1994)

[15] S. Ceri, and J. Widom. Deriving Production Rules for Constraint Maintenance, In: *Proc. VLDB, pp. 566-577*, (1990)

[16] R. Laleau, and F. Polack, Specification of integrity-preserving operations in information systems by using a formal UML-based language, In: *Information and Software Technology 43 pp. 693-704,* (2001).

[17] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. LöhrRichter, G. Saake, and H. Ehrich, Conceptual modelling of database applications using an extended ER model, In: *Data Knowledge Engineering 9, pp. 157-204,* (1992).

[18] D. Costal, M. Sancho, A. Olivé, and A. Roselló, The role of structural events in behaviour specification, In: *Proc. of DEXA'97, pp. 673-686,* (1997).

[19] James Welch, David Faitelson, and Jim Davies, Automatic maintenance of association invariants, In: *Software and Systems Modeling 7(3)*, (2008).

[20] Y. Ledru, Idenitfying pre-conditions with the Z/EVES theorem prover, In: *Proc. of the 13th International Conf. on Automated Software Engineering*, IEEE Computer Society Press, 1998.

[21] D. Costal, C. Gómez, A. Queralt, E. Teniente, Drawing Preconditions of Operation Contracts from Conceptual Schemas, In: *Proc. Of the Advanced Information Systems Engineering (CAiSE'08)*, (2008).

[22] Gentleware, Poseidon for UML, 14 Jan 2011, <http://www.gentleware.com/>

[23] Enterprise Architect, Sparx Systems, 14 Jan 2011, < http://www.sparxsystems.com.au/>

[24] Fujaba Tool Suit, Fujaba Project, 14 Jan 2011, <http://www.fujaba.de/>.

[25] Objecteering Software, Modelio, 14 Jan 2011, < http://www.modeliosoft.com/>

[26] IBM Software, Rational, 14 Jan 2011, <http://www-01.ibm.com/software/awdtools/developer/rose/>

[27] O. Pastor, E. Insfrán, V. Pelechano, J. Romero, and J. Merseguer, OO-METHOD: An OO software production environment combining conventional and formal methods, In: *Proc. of the 9th International Edition on Advanced Information Systems Engineering (CAiSE '97), pp. 145-158*, (1997).

[28] J. Cabot, R. Clarisó, and D. Riera, UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming, In: *Proc. of 22nd International Conference Automated Software Engineering (ASE'07), (2007).*

[29] M. Albert, Tratamiento de Asociaciones en Entornos de Producción Automática de Código, *Ph.D Thesis. Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.*

[30] A. J. McAllister, and D. Sharpe, An Approach for Decomposing N-Ary Data Relationships, In: *Software: Practice and Experience 28(2), pp. 125-154*, (1998).

[31] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual, (2005).

[32] S. Cook, and J. Daniels, Designing Objects Systems. Object-Oriented Modelling with Syntropy, *Prentice Hall*, (1994).

[33] D.F. D'Souza, and A.C. Wills, Catalysis- Objects, Components and Frameworks with UML, *Addison-Wesley, Reading, MA*, (1998).

[34]D. Firesmith, B. Henderson-Sellers, and I. Graham, The OML Reference Manual, *SIGS Books, NY,* (1997).

[35] F. Civello, Roles for composite objects in object-oriented analysis and design, In: *ACM SIGPLAN Notices 28(10)*, *pp. 376-393*, (1993).

[36] B. Henderson-Sellers, and F. Barbier, What Is This Thing Called Aggregation?, In: *J. Bosch R. Mitchell, A.C. Wills and B. Meyer, editors, Proc. of TOOLS 29*, *pp. 216-230, Los Alamitos, CA, USA,* (1999).

[37] J.J. Odell, Six different kinds of composition, In: *Journal of Object Oriented Programming (JOOP) 5(8)*, *pp. 10-15*, (1994).

[38] A.L. Opdahl, B. Henderson-Sellers, and F. Barbier, Ontological Analysis of Whole-Part Relationships in OO-models, In: *Information and Software Technology, 43 pp. 387-399*, (2001).

[39] M. Saksena, R.B. France, and M.M. Larrondo-Petrie, A Characterization of Aggregation, In: *C. Rollandand and G. Grosz, editors, Proc. of OOIS'98, pp. 11-19.,* (1998).

[40] Y. Wand, V.C. Storey, and R. Weber, An Ontological Analysis of the Relationship Construct in Conceptual Modeling, In: *ACM Transactions on Database Systems 24(4) pp. 494-528,* (1999).

[41] D. Milicév, On the Semantics of Associations and Association Ends in UML, In: *IEEE Transactions on Software Engineering 33(4),* (2007).

[42] J. Cabot, Incremental Integrity Checking in UML/OCL Conceptual Schemas, *PhD. Dissertation* (2006)*,* < http://jordicabot.com/papers/TesiJCabot.pdf>.

[43] ATLANMOD : Technologies de modélisation pour la production, le fonctionnement et l'évolution du logiciel, 14 Jan 2011, < http://www.inria.fr/recherche/equipes/atlanmod.fr.html >

[44] OO-Method Labs, 14 Jan 2011, < http://www.pros.upv.es/labs/index.php?option=com_content&view=category&id=21&Itemid=35&layout=default>

[45] B. Hetzel, The complete guide to software testing (2nd ed.), QED Information Sciences, Inc., Wellesley, MA, USA (1988).

[46] P. Giner, and V. Pelechano, Test-driven Development of Model Transformations, In: *Proc. of Model Driven Engineering Languages and Systems (Models'09), pp. 748-752,* (2009).

[47] A. Tort, OsCommerce conceptual schema, UPC, 2007.

[48] R. Raventós, A conceptual schema for a conference management application, UPC, Technical Report. 05-01-R, 2005.

[49] J. R. Abrial, The B-Book: Assigning Programs to Meanings, *Cambridge University Press,* (1996).

[50] S. Sendall, and A. Strohmeier, From Use Cases to System Operation Specifications, In: *Proc. of «UML» 2000 — the Unified Modeling Language, pp. 1-15,* (2000).

[51] D. Akehurst, G. Howells and K. McDonald-Maier. Implementing associations: UML 2.0 to Java 5, In: *Software and Systems Modeling 6 (1),* (2006).

[52] CARE Technologies, OLIVANOVA Programming Machine (ONME), 14 Jan 2011, <http://www.care-t.com/>

[53] F. S. Parreiras and S. Staab. Using ontologies with UML class-based modeling: The TwoUse approach. In: *Data Knowledge Engineering 69(11), pp. 1194-1207, November* (2010).

[54] I. Reinhartz-Berger. Towards automatization of domain modeling. In: *Data Knowledge Engineering 69(5) pp. 491-515, May* (2010).

[55] E.Planas, J. Cabot, and C.Gómez, Verifying Action Semantics Specifications in UML Behavioral Models, In: *Proc. Of the Advanced Information Systems Engineering (CAiSE'09)*, (2009).

Manoli Albert is Assistant Professor in the Department of Information Systems and Computation (DSIC) at the Universidad Politécnica de Valencia (Spain), where she is teaching Software Engineering and Design Patterns. She is a member of the the PROS Research Center at the UPV. She received her Ph.D. degree from the Valencia University of Technology in 2006. Her research interests are Model driven development, Conceptual Modelling, Requirements Engineering, Software Patterns and Ubiquitous and Pervasive systems.

Jordi Cabot received the BSc and PhD degrees in Computer Science from the Technical University of Catalonia. While working toward his PhD, he did a research stay at the Politecnico di Milano. He has held a senior lecturer position at the Open University of Catalonia and a postdoctoral fellow position at the University of Toronto and now he is an associate professor at the École des Mines de Nantes where he leads the AtlanMod INRIA/EMN research team. His research interests include conceptual modeling, model-driven development, formal verification and web engineering. He is a member of the IEEE and the ACM.

Cristina Gómez received her Degree in Informatics Engineering from the Universitat Politècnica de Catalunya in 1993. She later got his PhD degree from the same university in 2003. Currently, she is teaching Software Engineering at the Universitat Politècnica de Catalunya and at the Universitat Oberta de Catalunya. She is a member of the MPI research group at the Universitat Politècnica de Catalunya. Her research interest focuses on conceptual modeling, information systems and object-oriented analysis and design.

Vicente Pelechano is Associate Professor in the Department of Information Systems and Computation (DISC) at the Universidad Politécnica de Valencia, Spain. His research interests are Model Driven Development, Ubicomp and Ambient Intelligence, Web Engineering

and HCI. He received his Ph.D. degree from the Universidad Politécnica de Valencia in 2001. He is the head of the Ambient Intelligence and Web Technology Research Group in the ProS Research Center at the UPV. He has published in several well-known scientific journals, book chapters and international conferences. He is currently leading the technical supervision of the MOSKitt Open Source CASE Tool (http://www.moskitt.org).

Corresponding author: Manoli Albert

Complete Address:

Departamento de Sistemas y Computación

Universidad Politécnica de Valencia

Cami de Vera s/n 46022

Valencia, Spain

Tfn.: (+34) 96 387 7007 – Ext 83511

Fax: (+34) 96 3877359