

# From Declarative to Imperative UML/OCL Operation Specifications

Jordi Cabot

Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya  
Rbla. Poblenou 156. E08018 Barcelona, Spain  
jcabot@uoc.edu

**Abstract.** An information system maintains a representation of the state of the domain in its Information Base (IB). The state of the IB changes due to the execution of the operations defined in the behavioral schema. There are two different approaches for specifying the effect of an operation: the imperative and the declarative approaches. In conceptual modeling, the declarative approach is preferable since it allows a more abstract and concise definition of the operation effect and conceals all implementation issues. Nevertheless, in order to execute the conceptual schema, declarative specifications must be transformed into equivalent imperative ones.

Unfortunately, declarative specifications may be non-deterministic. This implies that there may be several equivalent imperative versions for the same declarative specification, which hampers the transformation process. The main goal of this paper is to provide a pattern-based translation method between both specification approaches. To facilitate the translation we propose some heuristics that improve the precision of declarative specifications and help avoid non-determinism in the translation process.

## 1 Introduction

A Conceptual Schema (CS) must include the definition of all relevant static and dynamic aspects of the domain [12]. Static aspects are collected in structural diagrams. Dynamic aspects are usually specified by means of a behavioral schema consisting of a set of system operations [14] (also known as domain events [19]) that the user may execute to query and/or modify the information modeled in the structural diagram. Without loss of generality, in this paper we assume that structural diagrams are expressed using object-oriented UML class diagrams [21] and that operations are specified in OCL [20].

There are two different approaches for specifying an operation effect: the *imperative* and the *declarative* approaches [28]. In an imperative specification, the conceptual modeler explicitly defines the set of structural events to be applied over the Information Base (IB). The IB is the representation of the state of the domain in the information system. A structural event is an elementary change (insertion of a new object, update of an attribute,...) over the population of the IB.

In a declarative specification, a contract for each operation must be provided. The contract consists of a set of pre and postconditions. A precondition defines a set of

conditions on the operation input and the IB that must hold when the operation is issued while postconditions state the set of conditions that must be satisfied by the IB at the end of the operation. In conceptual modeling, the declarative approach is preferable since it allows a more abstract and concise definition of the operation effect and conceals all implementation issues [28].

CSs must be executable in the production environment (either by transforming them into a set of software components or by the use of a virtual machine) [18]. To be executable, we must translate declarative behavior specifications into equivalent imperative ones.

The main problem hindering this translation is that declarative specifications are *underspecifications* [28] (also called non-deterministic), that is, in general there are several possible states of the IB that satisfy the postcondition of an operation contract. This implies that a declarative specification may have several equivalent imperative versions. We have a different version for each set of structural events that, given a state of the IB satisfying the precondition, evolve the IB to one of the possible states satisfying the postcondition.

The definition of a postcondition precise enough to characterize a single state of the IB is cumbersome and error-prone [4,26]. For instance, it would require specifying in the postcondition all elements not modified by the operation. There are other ambiguities too. Consider a postcondition as  $o.at_1 = o.at_2 + o.at_3$ , where  $o$  represents an arbitrary object and  $at_1$ ,  $at_2$  and  $at_3$  three of its attributes. Given an initial state  $s$  of the IB, states  $s'$  obtained after assigning to  $at_1$  the value of  $at_2 + o.at_3$  satisfy the postcondition. However, states where  $at_2$  is changed to hold the  $o.at_1 - o.at_3$  value or where, for instance, a zero value is assigned to all three attributes satisfy the postcondition as well. Strictly speaking, all three interpretations are correct (all satisfy the postcondition), though, most probably, only the first one represents the behavior the conceptual modeler meant when defining the operation.

In this sense, the main contribution of this paper is twofold:

1. We present several heuristics to clarify the interpretation of declarative operation specifications. We believe these heuristics represent usual modelers' assumptions about how the operation contracts should be interpreted when implementing the operations.
2. We define a set of patterns that use these heuristics in order to automatically translate an operation contract into a corresponding imperative operation specification.

As far as we know ours is the first method addressing the translation of UML/OCL operation contracts. Note that the high expressiveness of both languages increases the complexity of the translation process. We believe that the results of our method permit to leverage current model-driven development methods and tools by allowing code-generation from declarative specifications, not currently provided by such methods. Our translation is useful to validate the specification of the operations as well. After defining the operation contract, conceptual modelers could check if the corresponding imperative version reflects their aim and refine the contract otherwise.

The rest of the paper is organized as follows. Section 2 introduces the running example and some basic UML and OCL definitions. Section 3 presents our set of heuristics and Section 4 the list of translation patterns. Section 5 covers some inherently ambiguous declarative specifications. Section 6 sketches some implementation issues. Finally, Section 7 compares our approach with related work and Section 8 presents some conclusions and further research.

## 2 Running Example

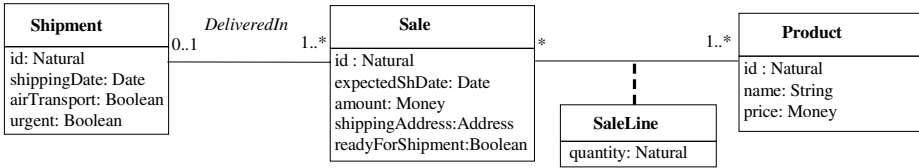
As a running example throughout the rest of the paper we will use the CS of Fig. 1 aimed at (partially) representing a simple e-commerce application. This CS is expressed by means of a UML class diagram [21]. Class diagrams consist of a set of classes (i.e. entity types) related by means of a set of associations (i.e. relationship types). Reified relationship types are called association classes in UML. Class instances are referred to as *objects* while association instances are known as *links*.

The CS contains information on sales (class *Sale*) and the products they contain (class *Product* and association class *SaleLine*). Sales are delivered in a single shipment (class *Shipment* and association *DeliveredIn*) but shipments may include several sales.

The CS includes also the contract of the *replanShipment* operation. This operation checks if shipments to be delivered soon have all their sales ready and replan them otherwise. The operation behavior is specified in OCL [20]. OCL is a formal high-level language used to write expressions on UML models. OCL admits several powerful constructs like iterators (*forAll*, *exists*, ...) and operations over collections of objects (*union*, *select*, ...). In OCL the implicit parameter *self* refers to the object over which the operation is applied. The dot notation is used to access the attributes of an object or to navigate from that object to the associated objects in a related class (the related class is identified by its role name in the association or the own class name when the name is not ambiguous).

For instance, in the precondition the expression *self.shippingDate* returns the value of the *shippingDate* attribute while *self.sale* returns the sales delivered in the shipment represented by the *self* variable. The *exist* iterator applied over this set of sales returns true if at least one sale satisfies the expression *not readyForShipment*.

In the postcondition we specify that there are two different ways of replanning the shipment depending on the value of the *urgent* attribute. We may either simply postpone the shipment until the new date given as an input (when it is not urgent) or to generate a new shipment to hold the sales that are not ready yet (and proceed with the usual shipment for the remaining ones). The expression *sh1.ocIsNew()* indicates that in the final state a new object (represented by the variable *sh1*) must exist and *sh1.ocIsTypeOf(Shipment)* indicates that this object must be instance of *Shipment*. The *includesAll* expression determines that shipment *sh1* must contain all non-ready sales (computed with the expression *self.sale@pre->select(not readyForShipment)*, where *@pre* indicates that *self.sale* is evaluated in the previous state, that is, in the state of the IB at the beginning of the operation execution), and so forth.



**context** Shipment::replanShipment(newDate:Integer)

**pre:** self.shippingDate>today() and self.shippingDate<today()+7 and self.sale->exists(not readyForShipment)

**post:** if self.urgent

then

sh1.ocIsNew() and sh1.ocIsTypeOf(Shipment) and sh1.shippingDate=newDate

and sh1.id=generateNewId() and sh1.airTransport=self.airTransport

and sh1.urgent=true and sh1.sale->includesAll(self.sale@pre->select(not readyForShipment))

and self.sale->excludesAll(self.sale@pre->select(not readyForShipment))

and sh1.sale->forAll(sl s.expectedShDate=sh1.shippingDate)

else self.shippingDate=newDate and self.sale->forAll(sl s.expectedShDate=self.shippingDate)

endif

**Fig. 1.** Our running example

### 3 Interpreting Declarative Specifications: A Heuristic Approach

Given the contract of an operation  $op$  and an initial state  $s$  of an IB (where  $s$  verifies the precondition of  $op$ ) there exist, in general, a set of final states  $set_s$  that satisfy the postcondition of  $op$ . All implementations of  $op$  leading from  $s$  to a state  $s' \in set_s$  must be considered correct. Obviously,  $s'$  must also be consistent with all integrity constraints in the schema, but, assuming a strict interpretation of operation contracts [24], the verification of those constraints need not to be part of the contract of  $op$ .

Even though, strictly speaking, all states in  $set_s$  are correct, only a small subset  $acc_s \subset set_s$  would probably be accepted as such by the conceptual modeler. The other states satisfy the postcondition but do not represent the behavior the modeler had in mind when defining the operation. In most cases  $|acc_s| = 1$  (i.e. from the modeler point of view there exists a single state  $s'$  that really “satisfies” the postcondition).

The first aim of this section is to detect some common OCL operators and expressions that, when appearing in a postcondition, increase the cardinality of  $set_s$ , that is, the expressions that cause an ambiguity problem in the operation contract. We also consider the classical *frame problem*, which, roughly, appears because postconditions do not include all necessary conditions to state which parts of the IB cannot be modified during the operation execution. Obviously, some of the problems could be avoided by means of reducing the allowed OCL constructs in the contracts but we will assume in the paper that this is not an acceptable solution.

Ideally, once the conceptual modeler is aware of the ambiguities appearing in an operation  $op$ , he/she should define the postcondition of  $op$  precise enough to ensure that  $acc_s = set_s$ . However, this would require specifying the possible state of every single object and link in the new IB state which is not feasible in practice [4,26].

Therefore, the second aim of this section is to provide a set of heuristics that try to represent common assumptions used during the specification of operation contracts. Each heuristic disambiguates a problematic expression  $exp$  that may appear in a

postcondition. The ambiguity is solved by providing a default interpretation for *exp* that identifies, among all states satisfying *exp*, the one that, most probably, represents what the modeler meant when defining *exp*.

With these heuristics, modelers do not need to write long and cumbersome postconditions to clearly specify the expected behavior of the operation. They can rely on our heuristic to be sure that, after the operation execution, the new state will be the one they intended. Our heuristics have been developed after analyzing many examples of operation contracts of different books, papers and, specially, two case studies ([11] and [23]) and comparing them, when available, with the operation textual description. Due to lack of space we cannot provide herein the list of examples we have examined.

In what follows we present our set of heuristics and discuss their application over our running example.

### 3.1 List of Heuristics

Each heuristic may target different OCL expressions. Note that other OCL expressions can be transformed into the ones tackled here by means of first preprocessing them using the rules presented in [8]. In the expressions, capital letters *X*, *Y* and *Z* represent arbitrary OCL expressions of the appropriate type (boolean, collection,...). The letter *o* represents an arbitrary object. The expression  $r_1.r_2\dots r_{n-1}.r_n$  represents a sequence of navigations through roles  $r_1..r_n$ .

#### Heuristic 1: Nothing else changes

- OCL expressions: –
- Ambiguity: Possible values for objects and links not referenced in the postcondition are left undefined.
- Default interpretation: Objects not explicitly referenced in the postcondition should remain unchanged in the IB (they cannot be created, updated or deleted during the transition to the new IB state). Links not traversed during the evaluation of the postcondition cannot be created nor deleted. Besides, for those objects that do appear in the postcondition, only those attributes or roles mentioned in the postcondition may be updated.

#### Heuristic 2: The order of the operands matters

- OCL expressions:  $X.a=Y$  (and in general any OCL binary operation)
- Ambiguity: There are three kinds of changes over an initial state resulting in a new state satisfying the above equality expression. We can either assign the value of expression *Y* to *a*, assign *a* to *Y* or assign to *a* and *Y* an alternative value *c*. In  $X.a$ , *a* represents an attribute or role of the objects returned by *X*.
- Default interpretation: In the new state *a* must have taken the value of *b*. Otherwise (that is, if the modeler's intention was to define that *b* should take the value of *a*) he/she would have most probably written the expression as  $Y.b = X.a$ . Note that if either operand is a constant value or is defined with the *@pre* operator (referring to the value of the operand in the previous state) just a possible final state exists because the only possible change is to assign its value

to the other operand (as usual, we assume that the previous state cannot be modified). This applies also to other ambiguities described in this section.

### Heuristic 3: Do not falsify the *if* clause

- OCL expressions: *if X then Y else Z* | *X implies Y*
- Ambiguity: Given an *if-then-else* expression included in a postcondition *p*, there are two groups of final states that satisfy *p*: 1 – States where the *if* and the *then* condition are satisfied or 2 – States where the *if* condition is false while the *else* condition evaluates to true. Likewise with expressions using *implies*.
- Default interpretation: To evaluate *X* and enforce *Y* or *Z* depending on the true value of *X*. Implementations of the operation that modify *X* to ensure that *X* evaluates to false are not acceptable (even if, for some states of the IB, it could be easier to falsify *X* in order to always avoid enforcing *Y*).

### Heuristic 4: Change only the last navigation in a navigation chain

- OCL expressions:  $X.r_1.r_2\dots r_{n-1}.r_n=Y$  (or any other operation over objects at  $r_n$ )
- Ambiguity: This expression may be satisfied in the final state by adding/removing links to change the set of objects obtained when navigating from  $r_{n-1}$  to  $r_n$ , by changing any intermediate role navigation  $r_i$  or by changing the value of *Y*.
- Default interpretation: To add/remove the necessary links on the association traversed during the last navigation ( $r_{n-1}.r_n$ ) in the navigation chain.

### Heuristic 5: Minimum insertions over the *includer* collection and no changes on the *included* one

- OCL expressions:  $X->includesAll(Y)$  |  $X->includes(o)$
- Ambiguity: All final states where, at least, the objects in *Y* (or *o*) have been included in *X* satisfy these expressions. However, states that, apart from those objects, add other objects to *X* also satisfy them as well as states where *Y* evaluates to an empty set (or *o* is null) since by definition all collections include the empty collection.
- Default interpretation: The new state *s'* should be obtained by means of adding to the initial state *s* the minimum number of links needed to satisfy the expression, that is, a maximum of  $Y->size()$  links must be created (just one for *includes* expressions). States including additional insertions are not acceptable and neither states where *Y* is modified to ensure that it returns an empty result.

### Heuristic 6: Minimum deletions from the *excluder* collection and no changes on the *excluded* one

- OCL expressions:  $X->excludesAll(Y)$  |  $X->excludes(o)$
- Ambiguity: All final states where, at least, the objects in *Y* (or *o*) have been removed from the collection of objects returned by *X* satisfy these expressions. However, states that, apart from those objects, remove other objects from *X* also satisfy them as well as states where *Y* evaluates to an empty set (or *o* is null) since then, clearly, *X* excludes all objects in *Y*.

- Default interpretation: The desired behavior is the one where the new state  $s'$  is obtained by means of removing from the initial state  $s$  the minimum number of links required to satisfy the expression and where  $Y$  has not been modified to ensure that it returns an empty set. Therefore, in  $s'$  a maximum of  $Y \rightarrow size()$  links may be deleted (or just one, for *excludes* expressions).

#### Heuristic 7: Do not empty the source collection of iterator expressions

- OCL expressions:  $X \rightarrow forAll(Y)$  (and, in general, all other iterator expressions)
- Ambiguity: There are two possible approaches to ensure that a *forAll* expression is satisfied in the new state of the IB. We can either ensure that all elements in  $X$  verify the  $Y$  condition or to ensure that  $X$  results in an empty collection since a *forAll* iterator over an empty set always returns *true*.
- Default interpretation: To ensure that all elements in  $X$  verify  $Y$  (and not to force  $X$  to be empty).

#### Heuristic 8: Minimum number of object specializations

- OCL expressions:  $o.oclIsTypeOf(Cl) \mid o.oclIsKindOf(Cl)$
- Ambiguity: These expressions require  $o$  to be instance of class  $Cl$  (or instance of a subtype of  $Cl$  when using *oclIsKindOf*). Therefore, new states where  $Cl$  is added to the list of classes that  $o$  is an instance of satisfy the expression. However, states where additional classes have been added or removed from  $o$  (assuming multiple classification) satisfy the expression as well.
- Default interpretation: The object  $o$  should only be specialized to  $Cl$  during the transition to the new state.

#### Heuristic 9: Minimum number of object generalizations

- OCL expressions:  $not\ o.oclIsTypeOf(Cl) \mid not\ o.oclIsKindOf(Cl)$
- Ambiguity: These expressions establish that, in the new state,  $o$  cannot be an instance of  $Cl$  (for *oclIsTypeOf* expressions) or an instance of  $Cl$  subtypes (for *oclIsKindOf* expressions). Therefore all states verifying this condition are valid even if they add/remove other classes from the list of classes where  $o$  belongs.
- Default interpretation: The object  $o$  should only be generalized to a supertype of  $Cl$ . If  $Cl$  has no supertypes,  $o$  must be completely removed from the IB.

### 3.2 Interpretation of *ReplanShipment* Using Our Heuristics

The expected behavior of *replanShipment* explained in Section 2 is just one of the (many) possible interpretations of *replanShipment* that satisfy its postcondition. Our heuristics prevent these alternative interpretations and ensure the described behavior.

As an example, heuristic 3 discards states where the value of the *urgent* attribute has been set to false (for instance, to avoid creating the new shipment), heuristic 2 ensures that variable *sh1* is initialized with the values of the *self* variable (and not the other way around), heuristic 7 discards states where the expression *self.sale*  $\rightarrow forAll$  is satisfied by means of removing all sales from *self* and so forth.

## 4 Patterns for a Declarative to Imperative Translation

Given a declarative specification of an operation  $op$  with a contract including a precondition  $pre$  and a postcondition  $post$ , the generated imperative specification for  $op$  follows the general form:

$$op(param_1 \dots param_n) \{ [ \text{if } pre \text{ then} ] \text{ translate}(post) [ \text{endif} ] \}$$

where  $translate(post)$  is the (recursive) application of our translation patterns over  $post$ . Testing the precondition is optional. Although usually added in object-oriented programming (*defensive programming* approach), it can be regarded as a redundant check [17] (the client should be responsible for calling  $op$  only when  $pre$  is satisfied).

The main purpose of our translation patterns is to draw from the postcondition definition a *minimal* set of structural events that, when applied over an initial state of the IB (that satisfies the precondition), reach a final state that verifies the postcondition. A set of structural events is minimal if no proper subset suffices to satisfy the postcondition [29].

When facing ambiguous OCL expressions, our patterns use the previous heuristics to precisely determine the characteristics of the desired final state and generate the needed structural events accordingly. This ensures that the final state, apart from satisfying the postcondition, is acceptable from the modeler's point of view. Getting rid of ambiguities also guarantees the determinism of the translation process.

As a result, the translation produces an imperative specification of the initial operation that could be used as an input for model-driven development tools in order to (automatically) generate its implementation in a given technology platform.

For the sake of simplicity we focus on the generation of the modifying structural events. We do not provide a translation for queries appearing in the postcondition into a set of primitive *read* events. Since queries do not modify the state of the IB, their translation is straightforward (and, in fact, most imperative languages for UML models allow expressing queries in OCL itself or in some similar language, see [16]).

For each pattern we indicate the OCL expression/s targeted by the pattern and its corresponding translation into a set of structural events. Our patterns do not address the full expressivity of the OCL but suffice to translate most usual OCL expressions appearing in postconditions. Additional OCL expressions can be handled with our method if they are first transformed (i.e. *simplified*) into equivalent OCL expressions (using the transformation rules presented in [8]) covered by our patterns.

### 4.1 Structural Events in the UML

The set of structural events allowed in UML behavior specifications is defined in the UML metamodel *Actions* packages [21] (structural events are called *actions* in the UML). The list of supported events<sup>1</sup> is the following:

- *CreateObject(Class c)*: It creates a new instance of  $c$ . This new instance is returned as an output parameter.
- *DestroyObject(Object o)*: It removes  $o$  from the IB. Optionally, we may indicate in the event that all links where  $o$  participated must be removed as well.

<sup>1</sup> For the sake of clarity, we distinguish between events over attributes and events over association ends (i.e. roles). UML unifies both concepts under the notion of *structural feature*.

- *AddAttributeValue(Attribute at, Object o, Object value)*: It adds *value* to the list of values for the attribute *at* of *o* (attributes may be multivalued in UML)
- *RemoveAttributeValue(Attribute at, Object o)*: It removes all values of attribute *at* in object *o*.
- *CreateLink(Association a, Object o<sub>1</sub>, ..., Object o<sub>n</sub>)*: It creates a new link for the association *a* relating objects *o<sub>1</sub>..o<sub>n</sub>*.
- *CreateLinkObject(Association a, Object o<sub>1</sub>, ..., Object o<sub>n</sub>)*: It creates a new link object (i.e. an association class instance) in *a* relating objects *o<sub>1</sub>..o<sub>n</sub>*.
- *DestroyLink(Association a, Object o<sub>1</sub>, ..., Object o<sub>n</sub>)*: It removes from *a* the link (or link object) between objects *o<sub>1</sub>..o<sub>n</sub>*.
- *RemoveLinkValue(AssociationEnd ae, Object o)*: It removes from *o* the values of the association end (i.e. role) *ae*. This event removes all links of the association *ae.association* (that returns, according to the UML metamodel, the association where *ae* belongs) where *o* participates.
- *ReclassifyObject(Object o, Class[] newClasses, Class[] oldClasses)*: It adds to the list of classes of *o* the classes specified in *newClasses* and removes those in *oldClasses*. Note that this event permits performing several generalizations and specializations at the same time.

**Table 1.** List of patterns. Column *N* indicates the pattern number. *Expression* describes the OCL expression targeted by each pattern and *Translation* the imperative code excerpt generated for it. In the patterns, *B<sub>i</sub>* stands for a boolean expression, *o* for an object variable and *X* and *Y* for two arbitrary OCL expressions of the appropriate type. *o.r* represents a navigation from *o* to the associated objects in the related class playing the role *r*.

N	Expression	Translation	Description
1	$B_1 \text{ and } \dots \text{ and } B_n$	<code>Translate(B<sub>1</sub>);</code> ... <code>Translate(B<sub>n</sub>);</code>	A set of boolean expressions linked by ANDs are transformed by translating each single expression sequentially.
2	$\text{if } B_1 \text{ then } B_2 \text{ else } B_3$	<code>if B<sub>1</sub> then Translate(B<sub>2</sub>);</code> <code>else Translate(B<sub>3</sub>);</code>	We translate both <i>B<sub>2</sub></i> and <i>B<sub>3</sub></i> and execute them depending on the evaluation of <i>B<sub>1</sub></i> (according to heuristic 3, a translation trying to falsify <i>B<sub>1</sub></i> is not acceptable).
3	$o.at=Y$ (where <i>at</i> is a univalued attribute)	<code>RemoveAttributeValue(at,o);</code> <code>AddAttributeValue(at,o,Y);</code>	We assign to the attribute <i>at</i> of <i>o</i> the value <i>Y</i> . The previous value is removed. Following heuristic 2, <i>Y</i> cannot be modified.
4	$o.at=Y$ (where <i>at</i> is multivalued)	<code>RemoveAttributeValue (at,o);</code> <code>foreach val in Y do</code> <code>AddAttributeValue(at,o,val);</code> <code>endfor;</code>	First, all previous values of <i>o.at</i> are removed. Then we assign one of the values of <i>Y</i> to each slot of <i>at</i> .
5	$o.r = Y$ (where <i>r</i> is a role with a '1' max multiplicity)	<code>RemoveLinkValue(r,o);</code> <code>CreateLink(r.association, o,Y);</code>	A new link relating <i>o</i> and <i>Y</i> in the association <i>r.association</i> is created ( <i>r.association</i> retrieves the association where <i>r</i> belongs to).
6	$o.r=Y$ (when <i>o.r</i> may return many objects)	<code>RemoveLinkValue (r,o);</code> <code>foreach o' in Y do</code> <code>CreateLink(r.association,o,o');</code> <code>endfor;</code>	We create a new link between <i>o</i> and each object in <i>Y</i> .

**Table 1.** (continued)

8	$X \rightarrow \text{forAll}(Y)$	<i>foreach</i> $o$ in $X$ <i>do</i> <i>if not</i> $(o.Y)$ <i>then</i> $\text{Translate}(o.Y)$ <i>endif</i> ; <i>endfor</i> ;	We ensure that each element affected by the <i>forAll</i> iterator verifies the $Y$ condition. According to heuristic 7, objects included in $X$ cannot be removed.
9	$o.\text{oclIsNew}()$ and $o.\text{oclIsTypeOf}(Cl)$	$o := \text{CreateObject}(Cl);$	The translation creates a new object of type $Cl$ . This new object is stored in the original postcondition variable. If $Cl$ is an association class $\text{CreateLinkObject}$ is used instead.
10	$o.\text{oclIsNew}()$ and $Cl.\text{allInstances}() \rightarrow \text{includes}(c)$		
11	<i>not</i> $o.\text{oclIsTypeOf}(OclAny)$	$\text{DestroyObject}(o);$	$o$ is deleted from the IB. This event deletes also all links where $o$ participates. ( $OclAny$ is the common supertype of all classes in an UML model).
12	<i>not</i> $o.\text{oclIsKindOf}(Cl)$ ( $Cl$ has no supertypes)		
13	$Cl.\text{allInstances}() \rightarrow \text{excludes}(o)$ ( $Cl$ has no supertypes)		
14	$o.\text{oclIsTypeOf}(Cl)$	$\text{ReclassifyObject}(o, Cl, Cl, \text{generalization.specific});$	The class $Cl$ is added to $o$ . Moreover, if $Cl$ has subtypes (retrieved using the navigation <i>generalization.specific</i> of the UML metamodel) these subtypes must be removed from $o$ ( $\text{oclIsTypeOf}$ is satisfied iff $Cl$ and the type of $o$ coincide).
15	$o.\text{oclIsKindOf}(Cl)$	$\text{ReclassifyObject}(o, Cl, []);$	$Cl$ is added to the list of classes of $o$ .
16	<i>not</i> $o.\text{oclIsTypeOf}(Cl)$ ( $Cl \ll OclAny$ )	$\text{ReclassifyObject}(o, [], Cl);$	$o$ is removed from $Cl$ but may remain instance of other classes in the model
17	<i>not</i> $o.\text{oclIsKindOf}(Cl)$ ( $Cl$ has supertypes)		
18	$Cl.\text{allInstances}() \rightarrow \text{excludes}(o)$ ( $Cl$ has supertypes)		
19	$o.r \rightarrow \text{includesAll}(Y)$	<i>foreach</i> $o'$ in $Y$ <i>do</i> $\text{CreateLink}(r.association, o, o')$ <i>endif</i> ;	A new link is created between $o$ and each object in $Y$ . If $o.r$ is a navigation towards an association class, $\text{CreateLinkObject}$ is used instead.
20	$o.r \rightarrow \text{includes}(Y)$	$\text{CreateLink}(r.association, o, Y);$	A link is created between $o$ and the single object returned by $Y$
21	$o.r \rightarrow \text{excludesAll}(Y)$	<i>foreach</i> $o'$ in $Y$ $\text{DestroyLink}(r.association, o, o')$ <i>endif</i> ;	All links between $o$ and the objects in $Y$ are destroyed.
22	$o.r \rightarrow \text{excludes}(Y)$	$\text{DestroyLink}(r.association, o, Y)$	The link between $o$ and the object in $Y$ is removed.
23	$o.r \rightarrow \text{isEmpty}(Y)$	<i>foreach</i> $o'$ in $o.r@pre$ $\text{DestroyLink}(r.association, o, o')$ <i>endif</i> ;	All links between $o$ and the objects returned by $o.r$ in the previous state are removed.

## 4.2 List of Patterns

Table 1 presents our list of translation patterns. The translation is expressed using a simple combination of OCL for the query expressions, the above structural events and, when necessary, conditional and iterator structures.

## 4.3 Applying the Patterns

Fig. 2 shows the translation of the *replanShipment* operation (Fig. 1). Next to each translation excerpt we show between brackets the number of the applied pattern.

```

context Shipment::replanShipment(newDate:Date)
{
if self.shippingDate>today() and self.shippingDate<today()+7 and self.sale->exists(not readyForShipment)
then
  if self.urgent (2)
  then (1)
    sh1:=CreateObject(Shipment); (9)
    AddAttributeValue(shippingDate, sh1, newDate); AddAttributeValue(id, sh1, generateNewId()); (3)
    AddAttributeValue(airTransport, sh1, self.airTransport); AddAttributeValue(urgent, sh1, true); (3)
    foreach o in self.sale@pre->select(not readyForShipment) CreateLink(DeliveredIn,sh1,o); endfor; (19)
    foreach o in self.sale@pre->select(not readyForShipment) DestroyLink(DeliveredIn, self, o); endfor; (21)
    foreach o in sh1.sale (8)
      if not o.expectedShDate=sh1.shippingDate
      then AddAttributeValue(expectedShDate,o,sh1.shippingDate); (3) endif;
    endfor;
  else
    AddAttributeValue(shippingDate,self,newDate); (3)
    foreach o in self.sale (8)
      if not o.expectedShDate=self.shippingDate
      then AddAttributeValue(expectedShDate,o,self.shippingDate); (3) endif;
    endfor;
  endif;
endif;
}

```

Fig. 2. Imperative version of *replanShipment*

## 5 Translating Inherently Ambiguous Postconditions

In some sense, all postconditions can be considered ambiguous. However, for most postconditions, the heuristics provided in Section 3 suffice to provide a single interpretation for each postcondition.

Nevertheless, some postconditions are inherently ambiguous (also called non-deterministic [2]). We cannot define heuristics for them since, among all possible states satisfying the postcondition, there does not exist a state clearly more appropriate than the others. As an example assume a postcondition including an expression  $a > b$ . There is a whole family of states verifying the postcondition (all

states where  $a$  is greater than  $b$ ), all of them equally correct, even from the modeler point of view or, otherwise, he/she would have expressed the relation between the values of  $a$  and  $b$  more precisely (for instance saying that  $a=b+c$ ).

We believe it is worth identifying these inherent ambiguous postconditions since most times the conceptual modeler does not define them on purpose but by mistake. Table 2 shows a list of expressions that cause a postcondition to become inherently ambiguous. We also provide a default translation for each expression so that our translation process can automatically translate all kinds of postconditions. Nevertheless, for these expressions user interaction is usually required to obtain a more accurate translation since the default translation may be too restrictive. For instance, for the second group of ambiguous expressions, the user may want to provide a specific constant value instead of letting the translation tool to choose an arbitrary one.

**Table 2.** List of inherently ambiguous expressions and their possible translation

Expression	Ambiguity description	Default Translation
$B_1$ or ... or $B_n$	At least a $B_i$ condition should be true but it is not defined which one(s)	To ensure that $B_i$ is true
$X <> Y$ , $X > Y$ , $X >= Y$ , $X < Y$ , $X <= Y$	The exact relation between the values of $X$ and $Y$ is not stated	To assign to $X$ the value of $Y$ plus/less a constant value of the appropriate type
$X+Y=W+Z$ (likewise with -, *, /, ...)	The exact relation between the values of the different variables is not stated	To translate the expression $X = W+Z-Y$
$X->exists(Y)$	An element of $X$ must verify $Y$ but it is not defined which one	To force the first element of $X$ to verify $Y$ (a total order relation must exist)
$X->any(Y)=Z$	Any element of $X$ verifying $Y$ could be the one equal to $Z$	To assign the value of $Z$ to the first element of $X$ verifying $Y$
$X.at->sum()=Y$	There exist many combinations of single values that once added result in $Y$	To assign to each object in $X$ a value $Y/X->size()$ in its attribute $at$
$X->asSequence()$	There are many possible ways of transforming a collection of elements $X$ into an (ordered) sequence of elements	Order in the sequence follows the total order of the elements in $X$ (a total order relation on $X$ must exist)
$X.r->notEmpty()$	The condition states that the navigation through the role $r$ must return at least an object but it is not stated how many nor which ones.	To assign a single object. The assigned object will be the first object in the destination class (a total order relation on the destination class must exist)
$op_1() = op_2()$	The return value of $op_1$ and $op_2$ must coincide. Depending on their definition several alternatives may exist.	Application of previous patterns depending on the specific definition of each operation

## 6 Tool Implementation

A prototype implementation of the translation presented in this paper has been developed. Given the XMI file representing the CS and the set of OCL operation

contracts in a textual form (parsed using the Dresden OCL toolkit [9]), the prototype translates the selected operations.

More specifically, the translation is obtained by means of traversing in preorder the OCL binary tree resulting from representing the OCL postcondition as an instance of the OCL metamodel [20]. For each tree node (where each node represents an atomic subset of the OCL expression: an operation, a constant, an access to an attribute, etc), the prototype chooses and applies the appropriate pattern. The complexity of the translation process is  $O(\log n)$ , being  $n$  the number of nodes of the tree.

Due to lack of space we cannot show this tree representation nor the details of the preorder traversal algorithm actually performing the translation.

## 7 Related Work

Two kinds of related work are relevant here: approaches devoted to the problem of improving the precision of declarative specifications (Section 7.1) and model-driven development methods and tools that may include facilities for generating code from operation contracts (Section 7.2).

### 7.1 Methods to Interpret Declarative Specifications

Methods aimed at disambiguating declarative specifications can be classified in three main groups: (1) methods that extend the contract with additional information, (2) methods that add implicit semantics to the contract expressions and (3) methods that try to characterize all possible new states satisfying the contract postcondition and let the modeler choose the one he/she prefers. This latter group (see [27] and [22] as examples) is not so well-explored and, currently, no method exists that is able to handle contracts defined in a language as expressive as the OCL.

Regarding the first group of methods, several formal languages (such as Z, VDM or JML) force the conceptual modeler to define in the contracts a new clause indicating which objects and links cannot change during the operation execution (*frame axioms*). [13] adapts the notion of frame axioms to OCL contracts. [4] uses a slightly different approach and asks modelers to specify which operations could have effected a change to a particular element. Other approaches, such as [2], combine the OCL with imperative extensions to clarify the semantics of the contract. The main limitations of all these approaches are: (1) they burden the modeler with the task of defining additional information in the contracts, (2) the addition of new elements to the structural diagram may require changing the frame axioms (now there are more elements that “cannot change”) and (3) the high expressiveness of the OCL limits their feasibility (for instance, postconditions may state, both, additions and removals over the set of objects returned by a navigation; it is not clear how frame axioms could be used to deal with this situation).

These problems can be avoided when adding implicit semantics to the expressions appearing in a postcondition, as we do in our heuristics proposal. We are aware that our heuristics require some strong assumptions about how the postconditions are specified, yet we believe the assumptions reflect the way conceptual modelers tend to (unconsciously?) specify the postconditions. We are not the first ones in proposing the

use of default semantics to simplify ambiguity problems of operation contracts. [4] recognizes that frame axioms could be (semi)automatically generated from the postcondition if assuming some implicit semantics. [26] proposes some basic assumptions regarding object (and collection) creations and removals. [6] proposes a *minimal change* heuristic (the preferred final state is the one with fewer changes wrt the initial one). However, this simple heuristic does not suffice to cover all possible ambiguities (see the different ambiguities commented in Section 3). Some ambiguous OCL expressions and their default interpretation were presented in a preliminary paper [7].

As a trade-off, this kind of methods requires modelers to agree in a given semantics when defining the contracts (either the ones we have assumed in our heuristics or alternative ones). We reckon that alternative approaches could be helpful when dealing with the inherently ambiguous postconditions of Section 5.

## 7.2 Methods for Code-Generation from Declarative Specifications

As far as we know, ours is the first approach to deal with the declarative-to-imperative translation of OCL operation specifications. Most methods and tools only support imperative specifications (see [16] as a representative example).

There exist several OCL tools allowing the definition of operation contracts (see, among others, [3,5,10,9]). However, during the code-generation phase, contracts are simply added as validation conditions. They are transformed into *if-then* clauses that check at the beginning and at the end of the operation if the pre and postconditions are satisfied (and raise an exception otherwise). The actual implementation of the operation must be manually defined. [1] checks the correctness of an implementation with respect to its contract but does not generate it.

A similar problem is faced in the database field when computing a sequence of updates that make the database to satisfy a given query (see [29] as an example). A typical example is the integrity maintenance problem (see [15] for a survey). Nevertheless, the limited expressivity of these methods (in terms of, both, the constraint definition language and the different types of structural events supported) prevents directly reusing them in the translation of UML/OCL operations.

## 8 Conclusion and Further Research

We have proposed a new method to transform an operation contract (declarative specification) into a set of structural events (imperative specification). The transformation process uses several heuristics that help draw the events from the OCL expressions included in the contract whenever their interpretation may be ambiguous.

Our translation may be useful to leverage current model-driven development tools, which up to now only support code-generation from imperative specifications. It may also be helpful for validation purposes, since modelers could immediately check which would be the implementation of their declarative specifications.

Our translation process has been validated against two case studies of real-life applications, a Car Rental System [11] and an e-marketplace system [23] as well as with other examples appearing in different books, papers and tutorials. Our patterns

have proven to be complete enough to translate most of the examples. Moreover, during the analysis we have detected several inherently ambiguous postconditions. In most cases, and according to the contract information in natural language, the original modelers were unaware of such ambiguities. We believe this is an additional benefit of applying our method.

As a further work, we plan to extend our translation process by combining the basic patterns presented up to now (this has been the main flaw of the method detected during its validation) and by considering the integrity constraints in the generation process to ensure that the generated implementation is consistent with the constraints and, at the same time, that the operation effect is preserved [25]. We are also interested in studying the applicability of our method in the reverse process, that is, in the translation from imperative to declarative specifications. Finally, we plan to work on the integration of our results and our prototype within an existing model-driven development tool.

## Acknowledgements

Thanks to the anonymous referees and the people of the GMC group (especially to Anna Queralt) for their useful comments to previous drafts of this paper. This work was partially supported by the Ministerio de Ciencia y Tecnologia and FEDER under project TIN2005-06053.

## References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool, Integrating object oriented design and formal verification. *Software and Systems Modeling* 4, 32–54 (2005)
2. Baar, T.: OCL and Graph-Transformations - A Symbiotic Alliance to Alleviate the Frame Problem. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 20–31. Springer, Heidelberg (2006)
3. Babes-Bolyai. Object Constraint Language Environment 2.0, <http://lci.cs.ubbcluj.ro/ocle/>
4. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering* 21, 785–798 (1995)
5. Borland. Borland® Together® Architect (2006)
6. Broersen, J., Wieringa, R.: Preferential Semantics for Action Specifications in First-order Modal Action Logic. In: *Proc. of the ECAI'98 Workshop on Practical Reasoning and Rationality* (1998)
7. Cabot, J.: Ambiguity issues in OCL postconditions. In: *Proc. OCL for (Meta-) Models in Multiple Application Domain (workshop co-located with the MoDELS'06 Conference)*, Technical Report, TUD-FI06-04-Sept (2006)
8. Cabot, J., Teniente, E.: Transformation Techniques for OCL Constraints. *Science of Computer Programming* (to appear), Available online: <http://dx.doi.org/10.1016/j.scico.2007.05.001>
9. Dresden. Dresden, OCL Toolkit, <http://dresden-ocl.sourceforge.net/index.html>
10. Dzidek, W.J., Briand, L.C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 10–19. Springer, Heidelberg (2006)

11. Frias, L., Queralt, A., Olivé, A.: EU-Rent Car Rentals Specification. LSI Technical Report, LSI-03-59-R (2003)
12. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base (1982)
13. Kosuczenko, P.: Specification of Invariability in OCL. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 676–691. Springer, Heidelberg (2006)
14. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice-Hall, Englewood Cliffs (2001)
15. Mayol, E., Teniente, E.: A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In: Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., Métais, E. (eds.) ER 1999. LNCS, vol. 1727, pp. 62–73. Springer, Heidelberg (1999)
16. Mellor, S.J., Balcer, M.J.: Executable UML. Object Technology Series. Addison-Wesley, London, UK
17. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
18. Olivé, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 1–15. Springer, Heidelberg (2005)
19. Olivé, A., Raventós, R.: Modeling events as entities in object-oriented conceptual modeling languages. *Data Knowl. Eng.* 58, 243–262 (2006)
20. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14)
21. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02)
22. Penny, D.A., Holt, R.C., Godfrey, M.W.: Formal Specification in Metamorphic Programming. In: Prehn, S., Toetenel, H. (eds.) VDM 1991. LNCS, vol. 551, pp. 11–30. Springer, Heidelberg (1991)
23. Queralt, A., Teniente, E.: A Platform Independent Model for the Electronic Marketplace Domain. LSI Technical Report, LSI-05-9-R (2005)
24. Queralt, A., Teniente, E.: Specifying the Semantics of Operation Contracts in Conceptual Modeling. *Journal on Data Semantics VII*, 33–56 (2006)
25. Schewe, K.-D., Thalheim, B.: Towards a theory of consistency enforcement. *Acta Informatica* 36, 97–141 (1999)
26. Sendall, S., Strohmeier, A.: Using OCL and UML to Specify System Behavior. In: Object Modeling with the OCL, The Rationale behind the Object Constraint Language, pp. 250–280. Springer, Heidelberg (2002)
27. Wahls, T., Leavens, G.T., Baker, A.L.: Executing Formal Specifications with Concurrent Constraint Programming. *Autom. Softw. Eng.* 7, 315–343 (2000)
28. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys* 30, 459–527 (1998)
29. Wüthrich, B.: On Updates and Inconsistency Repairing in Knowledge Bases. In: Proc. 9th Int. Conf. on Data Engineering, pp. 608–615 (1993)