Reasoning on UML Conceptual Schemas with Operations

Anna Queralt and Ernest Teniente

Universitat Politècnica de Catalunya
{aqueralt,teniente}@lsi.upc.edu

Abstract. A conceptual schema specifies the relevant information about the domain and how this information changes as a result of the execution of operations. The purpose of reasoning on a conceptual schema is to check whether the conceptual schema is correctly specified. This task is not fully formalizable, so it is desirable to provide the designer with tools that assist him or her in the validation process. To this end, we present a method to translate a conceptual schema with operations into logic, and then propose a set of validation tests that allow assessing the (un)correctness of the schema. These tests are formulated in such a way that a generic reasoning method can be used to check them. To show the feasibility of our approach, we use an implementation of an existing reasoning method.

Keywords: Conceptual modeling, automatic reasoning, operation contracts.

1 Introduction

The correctness of an information system is largely determined during requirements specification and conceptual modeling, since errors introduced at these stages are usually more expensive to correct than those made during design or implementation. Thus, it is desirable to detect and correct errors as early as possible in the software development process. Moreover, this is one of the key problems to solve for achieving the goal of automating information systems building [15].

The correctness of a conceptual schema can be seen from two different points of view. From an internal point of view, correctness can be determined by reasoning on the definition of the schema itself, without taking the user requirements into account. This is equivalent to answering to the question *Is the conceptual schema right?*. There are some typical properties that can be automatically tested to determine this kind of correctness like schema satisfiability, operation executability, etc.

On the other hand, from an external point of view, correctness refers to the accuracy of the conceptual schema regarding the user requirements [1] and it can be established by answering to the question *Are we building the right conceptual schema?*. Testing whether a schema is correct in this sense may not be completely automated since it necessarily requires the user intervention. Nevertheless, it is desirable to provide the designer with a set of tools that assist him during the validation process.

A *conceptual schema* consists of a *structural part*, which defines the relevant static aspects of the domain, and a *behavioral part*, which specifies how the information

P. van Eck, J. Gordijn, and R. Wieringa (Eds.): CAiSE 2009, LNCS 5565, pp. 47–62, 2009.

represented in the structural part changes as a result of the execution of system operations [11]. System operations specify the response of the system to the occurrence of some event in the domain, viewing the system as a black box and, thus, they are not assigned to classes. They define the only changes that can be performed on the IB.

Figs. 1 and 2 show a possibly incorrect conceptual schema of a (simplified) on-line auction site that we will use as a running example.

A test that the designer can perform to validate the internal correctness of the structural schema is to check whether it is satisfiable, that is, if it accepts at least one instance satisfying all the constraints. In our example, the following instantiation: *"Mick is a registered user who owns a book, and bids 200$ for a bicycle, owned by Angie, who had set a starting price of 180$"* satisfies all the graphical and textual constraints, which demonstrates that the structural schema is satisfiable.

However, the fact that the structural part is satisfiable does not necessarily imply that the whole conceptual schema also is. That is, when we take into account that the only changes admitted are those specified in the behavioral schema, it may happen that the properties fulfilled by the structural schema alone are no longer satisfied.

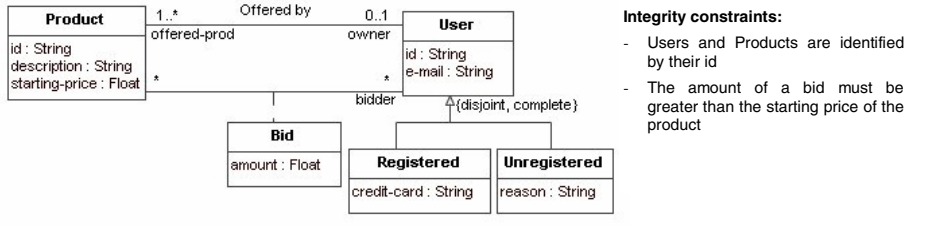

Fig. 1. The structural schema of an on-line auction site

```
Op:     registerUser(id: String, email: String, ccard: String)
Pre:
Post:   Registerd.allInstances()-> exists(u | u.oclIsNew() and
        u.id = id and u.email = email and u.credit-card = ccard)

Op:     unregisterUser(u: User, reason:String)
Pre:    u.oclIsTypeOf(Registered)
Post:   u.oclIsTypeOf(Unregistered) and u.reason = reason
        and not u.oclIsTypeOf(Registered)

Op:     placeBid(p: Product, u: User, value: Float)
Pre:    u.oclIsTypeOf(Registered)
Post:   Bid.allInstances()-> exists(b | b.oclIsNew() and b.user = u
        and b.product = p and b.amount = value)

Op:     offerProduct(u: User, id: String, desc: String, sp: Float)
Pre:    u.oclIsTypeOf(Registered)
Post:   Product.allInstances()->exists(p| p.oclIsNew() and p.id=id and
        p.description=desc and p.starting-price=sp and p.owner=u)
```

Fig. 2. A partial behavioral schema corresponding to the structural schema of Figure 1

In our example, although it is possible to find instances of *User* satisfying all the constraints as we have just seen, there is no operation that successfully populates this class. The operation *registerUser* seems to have this purpose but it never succeeds since it does not associate the new user with a *Product* by means of *Offered by*, which violates the cardinality constraint of the role *offered-prod*. As a consequence, since the only operation that creates a product (i.e. *offerProduct*) requires an existing user, there can not be any instance of *Product* either. Then, we have that this schema can never be populated using the operations defined and, although the structural part of the schema is semantically correct, the whole conceptual schema is not.

The main contribution of this work is to propose an approach to help to validate a conceptual schema with a behavioral part. To do this, we provide a method to translate a UML schema, with its behavioral part consisting of operations specified in OCL, into a set of logic formulas. The result of this translation is such that ensures that the only changes allowed are those specified in the behavioral schema, and can be validated using any existing reasoning method capable to deal with negation of derived predicates. To our knowledge, ours is the first approach that validates jointly the structural and behavioral parts of a UML/OCL conceptual schema.

We provide the designer with several validation tests which allow checking the correctness of a schema from the internal and external points of view mentioned above. Some of the tests are automatic and are directly generated from the conceptual schema while others are user-defined and give the designer the freedom to ask whichever questions he wants regarding situations that hold (do not hold) in the domain to ensure that they are (not) accepted by the schema. In both cases, the designer intervention is required to fix any problem detected by the tests.

We also show the feasibility of our approach by using an implementation of an existing reasoning method, which has had to be extended for our purposes.

Basic concepts are introduced in section 2. Section 3 presents our method to translate a schema with operations into logic. Section 4 presents our approach to validation. Section 5 shows its feasibility by means of an implementation. Section 6 reviews related work. Finally, we present our conclusions in section 7.

2 Basic Concepts

The *structural schema* consists of a taxonomy of entity types together with their attributes, a taxonomy of associations among entity types, and a set of integrity constraints over the state of the domain, which define conditions that each instantiation of the schema, i.e. each state of the information base (IB), must satisfy. Those constraints may have a graphical representation or can be defined by means of a particular general-purpose language.

In UML, a structural schema is represented by means of a class diagram, with its graphical constraints, together with a set of user-defined constraints, which can be specified in any language (Figure 1). As proposed in [21], we will assume these constraints are specified in OCL.

The content of the IB changes due to the execution of operations. The *behavioral schema* contains a set of *system operations* and the definition of their effect on the IB. System operations specify the response of the system to the occurrence of some event in the domain, viewing the system as a black box and, thus, they are not assigned to classes [11]. These operations define the only changes that can be performed on the IB.

An operation is defined by means of an operation contract, with a *precondition*, which expresses a condition that must be satisfied when the call to the operation is done, and a *postcondition,* which expresses a condition that the new state of the IB must satisfy. The execution of an operation results in a set of one or more *structural events* to be applied to the IB. Structural events are elementary changes on the content of the IB, that is, insertions or deletions of instances. We assume a strict interpretation of operation contracts [17] which prevents the application of an operation if any constraint is violated by the state satisfying the postcondition.

The operation contracts of the behavioral schema of our running example are shown in Figure 2. Each contract describes the changes that occur in the IB when the operation is invoked. Since we assume a strict interpretation, there is no need to include preconditions to guarantee the satisfaction of integrity constraints. However, if those preconditions were added, they would also be correctly handled by our method.

As we will see in Section 3, we translate a UML and OCL schema such as the one of the example into a set of first-order logic formulas in order to use a reasoning method to determine several properties on it. The OCL considered consists of all the OCL operations that result in a boolean value, including select and size, which can also be handled by our method despite returning a collection and an integer. The logic formalization of the schema consists of a set of rules and conditions defined as follows.

A *term* is either a variable or a constant. If p is a n-ary predicate and $T_1, \ldots, T_n$ are terms, then $p(T_1, \ldots, T_n)$ or $p(\overline{T})$ is an *atom.* An *ordinary literal* is either an atom or a negated atom, i.e. $\neg\ p(\overline{T})$. A *built-in literal* has the form of $A_1 \theta A_2$, where A_1 and A_2 are terms. Operator θ is either $<, \leq, >, \geq, =$ or $\neq$.

A normal clause has the form: $A \leftarrow L_1 \wedge \ldots \wedge L_m$ with $m \geq 0$, where A is an atom and each L_i is a literal, either ordinary or built-in. All the variables in A, as well as in each L_i, are assumed to be universally quantified over the whole formula. A is the *head* and $L_1 \wedge \ldots \wedge L_m$ is the *body* of the clause. A *fact* is a normal clause of the form $p(\bar{a})$, where $p(\bar{a})$ is a ground atom. A *deductive rule* is a normal clause of the form: $p(\overline{T}) \leftarrow L_1 \wedge \ldots \wedge L_m$ with $m \geq 1$, where p is the *derived* predicate defined by the deductive rule. A *condition* is a formula of the (*denial*) form: $\leftarrow L_1 \wedge \ldots \wedge L_m$ with $m \geq 1$.

Finally, a *schema S* is a tuple (*DR*, *IC*) where *DR* is a finite set of deductive rules and *IC* is a finite set of conditions. All these formulas are required to be *safe*, that is, every variable occurring in their head or in negative or built-in literals must also occur in an ordinary positive literal of the same body. An *instance of a schema S* is a tuple *(E,S)* where *E* is a set of facts about base predicates. *DR(E)* denotes the whole set of ground facts about base and derived predicates that are inferred from an instance *(E,S),* and corresponds to the fixpoint model of $DR \cup E$.

3 Translation of the Conceptual Schema into Logic

Validation tests that consider the structural schema alone are aimed at checking that an instantiation fulfilling a certain property and satisfying the integrity constraints can exist. In this case, classes, attributes and associations can be translated into base predicates that can be instantiated as desired, as long as integrity constraints are satisfied, in order to find a state of the IB that proves a certain property [16].

However, when considering also the behavioral schema, the population of classes and associations is only determined by the events that have occurred. In other words,

the state of the IB at a certain time *t* is just the result of all the operations that have been executed before *t,* since the instances of classes and associations cannot be created or deleted as desired. For instance, according to our schema in Fig. 1 and the operations defined, *Angie* may only be an instance of *Registered* at a time *t* if the operation *registerUser* has created it at some time before *t* and the operation *unregisterUser* has not removed it between its creation and *t.*

For this reason, it must be guaranteed that the population of classes and associations at a certain time depends on the operations executed up to that moment. To do this, we propose that operations are the basic predicates of our logic formalization, since their instances are directly created by the user. Classes and associations will be represented by means of derived predicates instead of basic ones, and their derivation rules will ensure that their instances are precisely given by the operations executed.

This approach clearly differs from our previous work [16, 18], where we proposed to formalize classes, attributes and associations as base predicates. Note, however, that a formalization of this kind does not ensure that instances of classes and associations result from the execution of operations.

3.1 Deriving Instances from Operations

Classes and associations are represented by means of derived predicates whose derivation rules ensure that their instances are given by the occurrence of operations, which are the base predicates of our formalization of the schema. Then, an instance of a predicate *p* representing a class or association exists at time *t* if it has been added by an operation at some time *t2* before *t*, and has not been deleted by any operation between *t2* and *t.* Formally, the general derivation rule is:

$$p([P,]P_1,...,P_n,T) \leftarrow addP([P,]P_1,...,P_n,T2) \wedge \neg deletedP(P_i,...P_j,T2,T) \wedge T2 \leq T \wedge time(T)$$
$$deletedP(P_i,...,P_j,T1,T2) \leftarrow delP(P_i,..,P_j,T) \wedge T > T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$$

where *P* is the OID (Object Identifier), which is included if *p* is a class. $P_i,...,P_j$ are the terms of *p* that suffice to identify an instance of *p* according to the constraints defined in the schema. In particular, if *p* is a class (or association class), $P=P_i=P_j$. The predicate *time* indicates which are the time variables that appear in the derived predicate we are defining. As well as those representing operations, *time* is a base predicate since its instances cannot be deduced from the rest of the schema. Predicates *addP* and *delP* are also derived predicates that hold if some operation has created or deleted an instance of *p* at time *T*, respectively. They are formalized as follows.

Let $op\text{-}addP_i$ be an operation of the behavioral schema, with parameters $Par_1,...,Par_n$ and precondition pre_i such that its postcondition specifies the creation of an instance of a class or association *p.* For each such operation we define the following rule:

$$addP([P,]Par_i,...,Par_k,T) \leftarrow op\text{-}addP_i([P,]Par_1,...,Par_m,T) \wedge pre_i(T_{pre}) \wedge T_{pre}=T\text{-}1 \wedge time(T)$$

where $Par_i,..,Par_k$ are those parameters of the operation that indicate the information required by the predicate *p,* and *T* is the time in which the operation occurs. The literal $pre_i(T_{pre})$ is the translation of the precondition of the operation, following the same rules used to translate OCL integrity constraints [16]. Note that, since the precondition must hold just before the occurrence of the operation, the time of all its facts is *T-1*.

Similarly, for each operation $op\text{-}delP_i(Par_1,...,Par_n,T)$ with precondition pre_i that deletes an instance of p we define the derivation rule:

$$delP(Par_i,...Par_j,T) \leftarrow op\text{-}delP_i(Par_1,...,Par_n,T) \wedge pre_i(T_{pre}) \wedge T_{pre}=T-1 \wedge time(T)$$

where $Par_i,...,Par_j$ are those parameters of the operation that identify the instance to be deleted. Thus, if p is a class or association class, *delP* will have a single term in addition to T, which corresponds to the OID of the deleted instance.

To completely define the above derivation rules for each predicate representing an element of the structural schema, we need to know which OCL operations of the behavioral schema are responsible for creating or deleting its instances. For our purpose, we assume that operations create instances with the information given by the parameters or delete instances that are given as parameters. A single operation can create and/or delete several instances. We are not interested in query operations since they do not affect the correctness of the schema.

Several OCL expressions can be used to specify that an instance exists or not at postcondition time. For the sake of simplicity, we consider a single way to specify each of these conditions, since other OCL expressions with equivalent meaning can be easily rewritten in terms of the ones we consider. Under this assumption, we define the rules to identify the creation and deletion of instances in OCL postconditions:

R1. An instance $c(I,A_1,...,A_n,T)$ of a class C is added by an operation if its postcondition includes the OCL expression: `C.allInstances()->exists(i| i.oclIsNew() and i.attr`$_i$`=a`$_i$`)` or the expression: `i.oclIsTypeOf(C) and i.attr`$_i$`=a`$_i$, where each $attr_i$ is a single-valued attribute of C.

R2. An instance $c(I,P_1,...,P_n,A_1,...,A_m,T)$ of an association class C is added by an operation if its postcondition includes the expression: `C.allInstances()-> exists(i| i.oclIsNew() and i.part`$_1$`=p`$_1$ `and...and i.part`$_n$`=p`$_n$ `and i.attr`$_1$`=a`$_1$ `and...and i.attr`$_m$`=a`$_m$`)` or the expression: `i.oclIsTypeOf(C) and i.part`$_1$`=p`$_1$ `and ... and i.part`$_n$`=p`$_n$ `and i.attr`$_1$`=a`$_1$ `and...and i.attr`$_m$`=a`$_m$, where each $part_i$ is a participant that defines the association class, and each $attr_j$ is a single-valued attribute of C.

R3. An instance $r(C_1,C_2,T)$ of a binary association R between objects C_1 and C_2, with roles *role-c*$_1$ and *role-c*$_2$ in r is added by an operation if its postcondition contains the OCL expression: `c`$_i$`.role-c`$_j$ `= c`$_j$, if the multiplicity of *role-c*$_j$ is at most 1 or the expression: `c`$_i$`.role-c`$_j$`-> includes(c`$_j$`)`, if the multiplicity of *role-c*$_j$ is greater than 1. This rule also applies to multi-valued attributes. Creation or deletion of instances of n-ary associations with n>2 cannot be expressed in OCL unless they are association classes, which are considered in the previous rule.

R4. An instance $c(I,A_1,...,A_n,T)$ of a class C is deleted by an operation if its postcondition includes the expression: `C`$_{gen}$`.allInstances()->excludes(i)` or the expression: `not i.oclIsTypeOf(C`$_{gen}$`)`, where C_{gen} is either the class C or a superclass of C.

R5. An instance $c(I,P_1,...,P_n,A_1,...,A_m,T)$ of an association class is deleted by an operation if its postcondition includes: `C.allInstances()-> excludes(i)`, or: `not i.oclIsTypeOf(C)`, or if any of its participants ($P_1,...,P_n$) is deleted.

R6. An instance $r(C_1,C_2,T)$ of a binary association R between objects C_1 and C_2, with roles *role-c*$_1$ and *role-c*$_2$ in r is deleted by an operation if its postcondition includes: `c`$_i$`.role-c`$_j$ `->excludes(c`$_j$`)` or if any of its participants (C_1 or C_2) is deleted.

For instance, according to the previous translation rules, the class *Registered* of our example will be represented by means of the clauses:

$$registered(U,Id,Email,Ccard,T) \leftarrow addRegistered(U,Id,Email,Ccard,T2) \wedge \neg deletedRegistered(U,T2,T) \wedge T2 \leq T \wedge time(T)$$
$$deletedRegistered(U,T1,T2) \leftarrow delRegistered(U,T) \wedge T > T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$$

where *U* corresponds to the unique OID required by every instance of a class. In turn, *addRegistered* and *delRegistered* are derived predicates whose definition depends on the operations of the behavioral schema that insert and delete instances of the class *Registered*. The operation *registerUser* creates an instance of *registered(U,Id,Email,C-card,T)* according to R1, since its postcondition includes the expression `Registered.allInstances()->exists(u| u.oclIsNew()and u.e-mail=e-mail and u.id=id and u.credit-card=ccard)`.. Since the other operations do not create instances of *Registered*, there is a single derivation rule for *addRegistered*:

$$addRegistered(U,Id,Email,Ccard,T) \leftarrow registerUser(U,Id,Email,Ccard,T) \wedge time(T)$$

We also need to find which operations are responsible for deleting instances of *Registered* in order to specify the derivation rule of *delRegistered.* The operation *unregisterUser* is the only one that deletes instances of *Registered* according to R4, since it includes the OCL expression `not u.oclIsTypeOf(Registered)`. Its postcondition also includes the creation of an unregistered user, but this will be taken into account when specifying the derivation rules of *addUnregistered* for predicate *unregistered.* This time the precondition is not empty, and requires that *u* is an instance of *Registered*, so the derivation rule in this case is:

$$delRegistered(U,T) \leftarrow unregisterUser(U,T) \wedge registered(U,Id,E,Cc,T_{pre}) \wedge T_{pre}=T-1 \wedge time(T)$$

Since a modification can be regarded as a deletion followed by an insertion, no specific derived predicates are needed to deal with them.

3.2 Constraints Generated

Since we assume that events cannot happen simultaneously, we need to define constraints to guarantee that two operations cannot occur at the same time. Constraints are expressed as formulas in denial form, which represent conditions that cannot hold in any state of the IB. Therefore, for each operation *o* with parameters $P_1,...,P_n$ we define the following constraint for each parameter P_i:

$$\leftarrow o(P_1 1,...,P_n 1,T) \wedge o(P_1 2,...,P_n 2,T) \wedge P_i 1 <> P_i 2$$

And for each pair *o, o2* of operations we define the constraint:

$$\leftarrow o(P_1,...,P_n,T) \wedge o2(Q_1,...,Q_m,T)$$

In our example, *unregisterUser(U,Reason,T)* requires the constraints:

$$\leftarrow unregisterUser(U,R,T) \wedge unregisterUser(U2,R2,T) \wedge U <> U2$$
$$\leftarrow unregisterUser(U,R,T) \wedge unregisterUser(U2,R2,T) \wedge R <> R2$$

and, for each other operation of the schema, a constraint like:

$$\leftarrow unregisterUser(U,R,T) \wedge registerUser(Id,Email,Ccard,T)$$

Moreover, the constraints of the UML structural schema are also translated into this kind of formulas. The set of constraints needed is exactly the one resulting from the translation of the structural schema [16], but now they are defined in terms of derived predicates instead of basic ones.

4 Our Approach to Validation

Our approach to validation is aimed at providing the designer with different kinds of tests that allow him to assess the correctness of the conceptual schema being defined. All of them take into account both the structural and the behavioral parts of the conceptual schema.

We express all tests in terms of checking the satisfiability of a derived predicate. So, for each validation test to be performed, a derived predicate (with its corresponding derivation rule) that formalizes the desired test is defined. With this input, together with the translated schema itself, any satisfiability checking method that is able to deal with negation of derived predicates can be used to validate the schema. We illustrate our approach using the translation of our example obtained as explained in Section 3.

4.1 Is the Conceptual Schema Right?

The tests devoted to check the internal correctness of the schema can be automatically defined, i.e. they can be performed without the designer intervention. Some of them correspond to well known reasoning tasks (such as schema satisfiability) while others refer to additional properties that can be automatically drawn from the conceptual schema and which are an original contribution of this paper.

4.1.1 Checking Strong Satisfiability

A schema is strongly satisfiable if there is at least one fully populated state of the IB satisfying all the constraints [12]. In the presence of operations, this means checking whether they allow creating at least a complete valid instantiation.

To perform this test, we need to define a derived predicate such that it is true when the schema is strongly satisfiable, i.e. if it is possible to have an instance of all classes and associations of the schema. In our example:

$$sat \leftarrow registered(U,Uid,Email,Ccard,T) \wedge unregistered(U2,Uid2,Email2,Reason,T) \wedge product(P,Pid,Descr,St\text{-}pr,T) \wedge bid(B,Prod,Bidder,Amt,T) \wedge offeredBy(P2,Owner,T)$$

As we discussed in the introduction, the schema of our example is not strongly satisfiable when the behavior of the operations is taken into account. To avoid this mistake, we may replace the original operation *registerUser* by the following one responsible for creating both an instance of *Registered* and an instance of a *Product* that will be offered by the new user when he is registered:

```
Op:     registerUser(id: String, email: String, ccard: String, pid:
            String, descr: String, st-price: Float)
Pre:
Post:   Registered.allInstances()->exists(u|u.oclIsNew() and u.e-mail
        = email and u.c-card=ccard and u.offered-prod->exists(p |
        p.oclIsNew() and p.id=pid and p.description=descr and
        p.starting-price=st-price))
```

Now, if we check satisfiability of the predicate *sat*, the answer is that the schema is strongly satisfiable. The following sample instantiation shows that all classes can be populated at time 4. It only includes instances of base predicates, since the derived ones can be obtained from them. Since our base predicates correspond to the operations, the sample instantiations obtained give a sequence of operation calls that leads to a state that is valid according to the schema:

{registerUser(john,john@upc.edu,111, p1, pen,10,1), unregisterUser(john,2),
registerUser(mary,mary@upc.edu,222,p2,pen,20,3), placeBid(mary,p1,25,4)}

That is, we need to register a new user *John* at time 1 and then unregister him to have an instance of *Unregistered*. After that, we create another user *Mary* to have an instance of *Registered*. Finally, to populate the class *Bid, Mary* bids for the pen *p1*.

4.1.2 Automatically Generated Tests

Following the ideas suggested by model-based testing approaches [20], there are some tests that can be automatically drawn from the concrete schema to be validated. As usual, they will help the designer to detect potentially undesirable situations admitted by the schema. Note, however, that we can already determine these situations at the conceptual schema level while, in general, model-based testing requires an implementation of the software system to execute the tests. The definition of an exhaustive list of such kind of tests is out of the scope of this paper.

For instance, in our example, although a product may have no owner according to the cardinality constraint 0..1 of *owner*, it will always have exactly one owner in practice with the given operations. This means that there is probably something that the designer overlooked when specifying the behavioral schema like an operation to allow users withdrawing offered products or that the cardinality constraint should be just 1.

The derivation rule that formalizes this situation, which can be automatically generated from the information provided by the conceptual schema, is the following:

$unownedProd \leftarrow product(P,Id,Descr,St\text{-}price,,T) \wedge \neg hasOwner(P,T)$
$hasOwner(P,T) \leftarrow offeredBy(P,Owner,T)$

The absence of a sequence of operations satisfiying *unownedProd* shows that the conceptual schema does not admit products without owner and, therefore, that the cardinality constraint of owner is not properly defined. We assume that the designer decides to define the cardinality constraint of *owner* to exactly 1 to fix this situation.

The general form of the previous test is as follows. If the predicate *minCardAssoc* is not satisfiable, it means that there is a potentially undesirable situation:

$minCardAssoc \leftarrow classJ(P_j,...,T) \wedge \neg hasAssoc(P_j,T)$
$hasAssoc(P,T) \leftarrow assoc([A], P_1,...,P_i,...,P_n,T)$

for each $j<>i$ representing a participant of *Assoc,* where P_i is the participant with minimum cardinality 0.

4.1.3 Testing Properties of the Operations

When dealing with operations additional validation tests can be performed, namely applicability and executability of each operation [4]. An operation is *applicable* if there is a state where its precondition holds. An operation is *executable* if it can be

executed at least once, i.e. if there is a state where its postcondition holds, together with the integrity constraints, and such that its precondition was also true in the previous state.

To illustrate these properties, let us consider an additional operation *removeProduct* to delete products:

```
Op:     removeProduct(p: Product)
Pre:    p.owner->isEmpty()
Post:   Product.allInstances()->excludes(p)
```

As can be seen, the precondition of this operation requires that the product being removed has no owner, which is not possible according to the cardinality constraint 1 of *owner*, just redefined in the previous section. This means that this operation is not applicable and the designer should avoid this situation by, for example, modifying the precondition.

The formalization for an operation O with precondition $pre(t)$ is:

$$applicable\_O \leftarrow pre(T)$$

If this *applicable_O* is not satisfiable, the operation is not applicable.

Although an operation is applicable, it may never be successfully executed because it always leaves the IB in an inconsistent state. For instance, let us consider an additional operation *removeUser* that deletes the specified user as long as he or she has not bidden for any product:

```
Op:     removeUser(u: User)
Pre:    u.bid->isEmpty()
Post:   User.allInstances()->excludes(u)
```

This operation is applicable, since its precondition can be satisfied, but the postcondition removes a user, which is necessarily the owner of some product according to the cardinality constraint 1..* of *offered-prod.* Since this operation does not remove the products offered by the user, the resulting state of the IB will always violate the cardinality constraint 1 of *owner* for all products offered by *u*. This means that the execution of this operation will always be rejected because it is impossible to satisfy its postcondition and the integrity constraints at the same time.

To check executability, an additional rule has to be added to the translation of the schema to record the execution of the operation. In this case, if *executed_O* is satisfiable, then O is executable:

$$executed\_O \leftarrow o(P_1,...,P_n,t) \wedge pre(T\text{-}1)$$

4.2 Is It the Right Conceptual Schema?

Once the internal correctness of the schema is ensured by the previous tests, the designer will need to check its external correctness, i.e. whether it satisfies the requirements of the domain.

Our approach allows testing whether a certain desirable state that the designer may envisage is acceptable or not according to the current schema. The designer may define such a state either by means of a set of instances that classes and associations should contain at least; or by a derived predicate that defines it declaratively. Once a test is executed, the designer should compare the obtained results to those expected

according to the requirements and apply modifications to the conceptual schema if necessary.

For instance, an interesting question could be *"May a user place a bid on a product he is offering?*." To test this situation, the designer should define the rule:

$$bidderAndOwner \leftarrow bid(B,Prod,Usr,Amt,T) \wedge offeredBy(Prod,Usr,T)$$

In this case, *bidderAndOwner* is satisfiable, as shown by the sample instantiation:

{registerUser(john, john@upc.edu, 111, prod1, pen, 10, 1), placeBid(john, prod1, 15, 3) }

This result might indicate that the conceptual schema should restrict a user to place a bid on the products he owns either by defining an additional constraint in the structural schema or by strengthening the precondition of the operation *placeBid.*

As mentioned above, the designer may also specify additional tests by giving several instances of classes and associations and check whether there is at least a state that contains them (probably in addition to other instances). As an example, the designer could wonder whether a certain user, e.g. *joan*, may place two bids for the same product, e.g. *book1*. This situation may be tested by determining whether there is a state that contains the instances *{bid(1,book1,joan,9,1), bid(5,book1,joan,25,7)},* obtaining a negative answer in this case since, according to the semantics of associations, two instances of *Bid* cannot be defined by the same instances of *User* and *Product* .Since a user should be able to rebid for a product, this schema is not correct and should be modified by changing the definition of *Bid.*

By studying the results of the previous tests, and with his knowledge about the requirements of the system to be built, the designer will be able to decide if the schema is correct, and perform the required changes if not.

Checking the external correctness of a schema can also be partially automated by generating additional tests that check other kinds of properties. For instance, given a recursive association *Assoc*, it may be interesting to check whether an instance of the related class can be associated to itself. If the predicate *assocHasCycles* is satisfiable, then a constraint to guarantee that the association is acyclic or irreflexive, as it is usual in practice, may be missing:

$$assocHasCycles \leftarrow assoc(X,X,T)$$

5 Implementing Our Approach within an Existing Method

We have studied the feasibility of our approach by using an existing reasoning procedure, the CQC-Method [8], to perform the tests. To do this, we have extended a Prolog implementation of this method to incorporate a correct treatment of the time component of our atoms. We have executed this new implementation on our example to perform all validation tests that we have explained throughout the paper. We have also needed to implement the translation of the UML/OCL schema into logic.

The CQC Method is a semidecision procedure for finite satisfiability and unsatisfiability. This means that it always terminates if there is a finite example or if the tested property does not hold. However, it may not terminate in the presence of solutions with infinite elements. Termination may be assured by defining the maximum number of elements that the example may contain.

Roughly, the CQC Method is aimed at constructing a state that fulfills a goal and satisfies all the constraints in the schema. The goal to attain is formulated depending on the specific reasoning task to perform. In this way, the method requires two main inputs besides the conceptual schema definition itself. The *goal to attain*, which must be achieved on the state that the method will try to construct; and the set of *constraints to enforce*, which must not be violated by the constructed state.

Then, to check if a certain property holds in a schema, this property has to be expressed in terms of an initial goal to attain (G_0) and the set of integrity constraints to enforce (F_0), and then ask the CQC Method to attempt to construct a sample IB to prove that the initial goal G_0 is satisfied without violating any integrity constraint in F_0.

This means that, to perform our validation tests, we need to provide the CQC Method with the formalization of our schema, i.e. the derived predicates that represent classes and associations, the set of constraints of the schema as F_0 and the derived predicate formalizing the validation test to perform as G_0.

5.1 Variable Instantiation Patterns

The CQC Method performs its constraint-satisfiability checking tests by trying to build a sample state satisfying a certain condition. For the sake of efficiency the method tests only those variable instantiations that are relevant, without losing completeness. The method uses different *Variable Instantiation Patterns (VIPs)* for this purpose according to the syntactic properties of the schema considered in each test. The key point is that the VIPs guarantee that if the variables in the goal are instantiated using the constants they provide and the method does not find any solution, then no solution exists.

The VIP in which we are interested is the *discrete order VIP*. In this case, the set of constants is ordered and each distinct variable is bound to a constant according to either a former or a new location in the total linear order of constants maintained. The value of new variables is not always *static* (i.e. a specific numeric value), it can be a relative position within the linear ordering of constants. These are called *virtual constants*. For instance, in the ordering of constants {1, *d*, 6}, *d* is a virtual constant such that $1<d<6$. Then, its possible absolute values are 2 to 5. It may happen that the goal succeeds or fails without the need for further instantiations, and in this case *d* will never be bound to a concrete value.

To correctly instantiate the variables representing occurrence times that we have introduced in our translation of the conceptual schema, it has been necessary to add a *temporal VIP*. This new VIP has some similarities with the *discrete order VIP,* since they both deal with discrete values, order comparisons and negation, but it extends it to be able to bind a constant, either virtual or static, with its immediate successor. This is needed because our derivation rules require that preconditions hold exactly in the time immediately previous to the postcondition, not at any time before the postcondition. Then, we use a separate set of constants, with its own ordering, to deal with variables representing event times and we instantiate them with our *temporal VIP*.

For instance, assume we are attempting to derive an *Unregistered* user which must hold at time *d,* being *d* a virtual constant and {1, *d,* 5} our set of temporal constants. According to the precondition of *unregisterUser*, the user must be registered at *d-1*. Thus, since $1<d<5$, the time variable of the corresponding instance of *Registered* must

be instantiated either with 1 or with a virtual constant *f, f=d-1*. So, the relevant sets of constants are {[1, *d*], 5} and {1,[*f*, *d*],5}, where constants between brackets are tied so that no new constant can be ever placed between them.

The *temporal VIP* is formalized as follows. A variable instantiation step performs a transition from ($T \varnothing KT_i$) to ($\varnothing \theta KT_{i+1}$) that instantiates the temporal variable T according to one of the VIP-rules, where θ is a ground substitution of T and KT_i is the set of temporal constants. Let d_i denote virtual constants, c_i denote static constants and k_i denote either static or virtual constants, and let G_c be the current goal. The *temporal VIP* consists of the VIP-rules of the *discrete order VIP*, extended by the following rules, that apply when instantiating a temporal constant T such that $T = k_i\text{-}1$, $k_i \in KT_i$:

Tmp1. $\theta = T/c_{prev}$ and $KT_{i+1}=KT_i$, where $c_{prev}=c_{suc}\text{-}1$, $\{c_{suc},c_{prev}\}\subseteq KT_i$, $\{T=c_{suc}\text{-}1\}\in G_c$

Tmp2. $\theta = T/k$ and $KT_{i+1} = KT_i$, where $\{k, k_{suc}\} \subseteq KT_i$, $\{T =k_{suc}\text{-}1\}\in G_c$, there is no constant k_{prev} such that $k<k_{prev}<k_{suc}$ and k is tied to k_{suc} in KT_{i+1}.

Tmp3. $\theta = T/c_{new}$ and $KT_{i+1} = KT_i \cup \{c_{new}\}$, where $c_{new}=c_{suc}\text{-}1$, $c_{new}\notin KT_i$, $c_{suc}\in KT_i$, $\{T=c_{suc}\text{-}1\}\in G_c$, there is no d_{prev} tied to c_{suc} in KT_i, and there is no $c_{prev} \in KT_i$ such that $c_{prev} < c_{suc}$ and $|\{d_i \mid d_i \in KT_i \text{ and } c_{prev} < d_i < c_{suc}\}| < |c_{suc} - c_{prev}| - 1$.

Tmp4. $\theta = T/d_{new}$ and $KT_{i+1} = KT_i \cup \{d_{new}\}$, where $d_{new}\notin KT_i$, $d_{suc}\in KT_i$, $\{T=d_{suc}\text{-}1\}\in G_c$, there is no d_{prev} tied to d_{suc} in KT_i, d_{new} is tied to d_{suc} in KT_{i+1} and there are no c_i, $c_j \in KT_i$ such that $c_i < d_{suc} < c_j$, there is no c_m with $c_i < c_m < c_j$ and $|\{d_i \mid d_i \in KT_i \text{ and } c_i < d_i < c_j\}| < |c_j - c_i| - 1$.

6 Related Work

In this section we focus on those approaches that deal with UML schemas with a behavioral part. Thus, we leave out from our comparison to previous work those approaches that only deal with the structural schema [10, 12, 16], since satisfiability of the structural part does not necessarily imply that the whole conceptual schema is also satisfiable; as well as the first proposals to deal with behavior, in the context of deductive conceptual schemas [4, 5].

Due to its relevance, and despite not dealing with UML schemas, we believe it is worth including the Alloy language and analyzer [14] in this comparison. Alloy provides interesting validation capabilities for expressive schemas by searching for examples of the tests specified by the designer. The preconditions and postconditions of the operations can be checked manually, before and after each execution.

One of the first approaches to check satisfiability of UML schemas with operations is [6]. General constraints are handled, but they must be expressed in Z instead of OCL, which is the language recommended by the UML to formalize constraints and operations. Besides checking satisfiability of the structural schema, operations to insert, delete and update the instances of each class or association are automatically generated.

An approach to reason on UML/OCL schemas is HOL-OCL [2]. The method uses a theorem prover to determine some properties, such as equivalence of two integrity constraints, or applicability and executability of operations.

Another interesting tool to validate UML/OCL conceptual schemas is USE [9], which allows to test if a given instantiation is accepted by the schema taking into account the OCL constraints. Preconditions and postconditions can also be validated, but the execution of the operation has to be simulated manually.

Recently, and also for UML/OCL schemas, [3] reports a set of properties regarding the correctness of operations such as applicability or executability.

All the previous approaches have an important common drawback. None of them takes into account the definition of operations when determining whether a state is accepted or not by the schema. This means that a state may be reported as valid when, in fact, it is impossible to construct using the operations defined. This also damages the results obtained when testing the applicability of operations, since the state that satisfies a precondition may not be obtained by means of the operations defined. In fact, all these approaches would give an incorrect answer to 5 out of the 6 properties tested in this paper.

One of the approaches that does not share this drawback is [7], which combines state and event-based descriptions of a system to enable the automatic verification of dynamic properties regarding the system behavior. It may handle UML class diagrams but assumes that the system behavior is specified in the B and CSP languages, instead of OCL, and it is mainly aimed at testing properties related to the correct sequencing of the operations specified in the conceptual schema.

The other approach that takes the operations into account when determining whether a state is accepted or not by the schema belongs to the Rodin project. It combines UML-B [19] and ProB [13], the former to represent the schema and translate it into the B language, and the latter to validate it by animation. However, UML-B only accepts a subset of the UML that is suitable for translation into B, and constraints and operations must be directly expressed in B by the designer. Additionally, ProB requires that the search space is made finite by enumerating the values to be used in the animation. Since the fact that a property does not hold for those values does not mean that it can never hold, completeness is not guaranteed.

Finally, all of the approaches are able to check either the internal correctness [3, 6, 13] or the external correctness of the schema [2, 9, 14], but not both as we do.

7 Conclusions and Further Work

We have proposed a new approach to validate a UML conceptual schema, with textual OCL constraints and operations. To our knowledge, ours is the first approach that validates jointly the structural and behavioral parts of a UML/OCL schema.

Our approach allows determining automatically whether the conceptual schema is correctly defined, through tests about the accomplishment of desirable properties; and provides also a help to the designer to check that the schema defined is the right conceptual schema in the sense that it correctly specifies the requirements.

This is achieved by translating the UML conceptual schema, including its behavioral part, into a logic representation which incorporates the effect of operations in terms of the instances of classes and associations that are created or deleted. In this way, we ensure that the only changes allowed are those defined in the behavioral schema. With this logic representation, we can formalize each validation test in terms of checking the satisfiability of a derived predicate. Then, any satisfiability checking method able to deal with negation of derived predicates can be used to validate the schema.

We have also shown the feasibility of our approach by using and extending an implementation of an existing reasoning procedure, called CQC-Method [8], and applying it to our running example.

There are some interesting directions for further work, like applying the decidability results of our previous work [18] to schemas with a behavioral part, extending our approach to validate conceptual schemas with derived UML information or investigating the applicability of this approach to large conceptual schemas.

Acknowledgements. This work has been partly supported by Ministerio de Ciencia y Tecnología under TIN2008-00444/TIN, Grupo Consolidado, and TIN2008-03863.

References

1. Adrion, W.R., Branstad, M.A., Cherniavsky, J.C.: Validation, Verification and Testing of Computer Software. ACM Comput. Surv. 14(2), 159–192 (1982)
2. Brucker, A.D., Wolff, B.: The HOL-OCL Book. Swiss Federal Institute of Technology (ETH), 525 (2006)
3. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL Operation Contracts. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 40–55. Springer, Heidelberg (2009)
4. Costal, D., Teniente, E., Urpí, T., Farré, C.: Handling Conceptual Model Validation by Planning. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 255–271. Springer, Heidelberg (1996)
5. Díaz, O., Paton, N.W., Iturrioz, J.: Formalizing and Validating Behavioral Models through the Event Calculus. Information Systems 23(3/4), 179–196 (1998)
6. Dupuy, S., Ledru, Y., Chabre-Peccoud, M.: An Overview of RoZ: A Tool for Integrating UML and Z Specifications. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 417–430. Springer, Heidelberg (2000)
7. Evans, N., Treharne, H., Laleau, R., Frappier, M.: Applying CSP ‖ B to Information Systems. Software and System Modeling 7, 85–102 (2008)
8. Farré, C., Teniente, E., Urpí, T.: Checking Query Containment with the CQC Method. Data and Knowledge Engineering 53(2), 163–223 (2005)
9. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based Specification Environment for Validating UML and OCL. Science of Computer Programming 69(1-3), 27–34 (2007)
10. Hartmann, S.: Coping with Inconsistent Constraint Specifications. In: Kunii, H.S., Jajodia, S., Sølvberg, A. (eds.) ER 2001. LNCS, vol. 2224, pp. 241–255. Springer, Heidelberg (2001)
11. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edn. Prentice Hall PTR, Englewood Cliffs (2004)
12. Lenzerini, M., Nobili, P.: On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. In: Proc. 13th International Conference on Very Large Databases - VLDB 1987, pp. 147–154 (1987)
13. Leuschel, M., Butler, M.: ProB: An Automated Analysis Toolset for the B Method. Software Tools for Technology Transfer (2008) DOI: s10009-007-0063-9
14. MIT. The Alloy Analyzer. MIT Software Design Group, `http://alloy.mit.edu`

15. 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)
16. Queralt, A., Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 497–512. Springer, Heidelberg (2006)
17. Queralt, A., Teniente, E.: Specifying the Semantics of Operation Contracts in Conceptual Modeling. In: Spaccapietra, S. (ed.) Journal on Data Semantics VII. LNCS, vol. 4244, pp. 33–56. Springer, Heidelberg (2006)
18. Queralt, A., Teniente, E.: Decidable Reasoning in UML Schemas with Constraints. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 281–295. Springer, Heidelberg (2008)
19. Snook, C., Butler, M.: UML-B: Formal Modeling and Design Aided by UML ACM Trans. on Soft. Engineering and Methodology 15(1), 92–122 (2006)
20. Utting, M., Legeard, B.: Practical Model-Based Testing. Morgan Kaufmann, San Francisco (2006)
21. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley Professional, Reading (2003)