Validating UML Schemas with OCL Constraints and Operations

Anna Queralt and Ernest Teniente
Universitat Politècnica de Catalunya
{queralt, 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 validating a conceptual schema is to check whether it specifies what the designer intended. This task is not fully formalizable, so it is desirable to provide the designer with a set of tools that assist him or her in the validation process. To this end, we present a method to translate a complete schema into logic, and then propose a set of validation tests that can be performed on the schema. These tests are formulated in such a way that any reasoning method can be used to check them. To show the feasibility of our approach, we use an implementation of an existing reasoning method.

1. Introduction

In software quality assurance, validation is the determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements [1]. Intuitively, the purpose of the validation process is to answer to the question Am I building the right system?.

While it is true that software quality is usually determined by means of the final product, there are other quality factors that do not only depend on the implementation. In fact, the quality of an information system is largely determined early in the development cycle, i.e. during requirements specification and conceptual modeling. Additionally, errors introduced at these stages are usually much more expensive to correct than errors introduced during design or implementation. Thus, it is desirable to prevent, detect and correct errors as early as possible in the development process. Moreover, this has been identified as one of the key problems to be solved for achieving the goal of automating information systems building [15].

In this context, validation can be used to assure the quality of a conceptual schema instead of a piece of code. In this case, it aims to guarantee that a conceptual schema properly reflects what the user needs from an application. Since these needs or requirements are not usually formalized, it is desirable to provide the designer with a set of tools that assist him or her in the validation process [4], so that he can check whether the conceptual schema properly specifies what he intended.

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
represented in the structural part changes as a result of the execution of operations. Those operations define the only possible ways users can interact with the system.

<table>
<thead>
<tr>
<th>Product</th>
<th>Offered by</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: String</td>
<td>owner: String</td>
<td>email: String</td>
</tr>
<tr>
<td>description: String</td>
<td>standing-price: Float</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bid</th>
<th>Registered</th>
<th>Unregistered</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount: Float</td>
<td>credit-card: String</td>
<td>reason: String</td>
</tr>
</tbody>
</table>

Integrity constraints:
- Users and Products are identified by their id
- The amount of a bid must be greater than the starting price of the product
- The owner of a product cannot bid for it

Fig. 1 shows the structural schema of a (simplified) on-line auction site that we will use as a running example. The system stores information about users, which can be registered or unregistered. A user is the owner of a set of products (at least one), and a product belongs to exactly one user. Users can bid for products by specifying the amount they offer for the products in which they are interested. This structural schema includes some constraints that must be satisfied, in addition to those graphically represented in the class diagram. These constraints state that users and products are identified by their id, that a bid for a product must be greater than its starting price, and that the owner of a product cannot bid for that product.

A test that the designer can perform to validate the schema is to check whether it is satisfiable, that is, if it accepts at least one instance satisfying all the constraints. For example, the following instantiation of the schema:

"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 of the schema, which demonstrates that the structural schema is satisfiable. However, the fact that the structural part of a conceptual schema 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. For instance, if the schema does not contain any operation that successfully populates the class User, the schema will not be satisfiable, since it is not possible to populate any other class (instances of product will neither exist, since each Product needs an owner and, in turn, bids need products and users). This means that the result of a validation test strongly depends on which operations are contained in the behavioral schema, and how they are defined.

Several proposals deal with the validation of the structural part of ER and UML conceptual schemas, determining for example its satisfiability [3, 11, 13, 16] or allowing the designer to check if a given state is consistent according to the constraints defined in the schema [10, 16]. However there are a few proposals that take the behavioral schema into account in the validation process [5, 6, 9, 14], none of them dealing with UML and OCL schemas.

There are three main contributions made by this work. Firstly, we propose an approach to validate a complete conceptual schema (including its 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 or
tool that is capable of dealing with negation of derived predicates. Secondly, we
provide an approach to validate both the structural and the behavioral parts of the
schema by means of a set of tests. Finally, we 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 the
validation of a schema. Section 5 shows the feasibility of the previous results by means
of an implementation. Section 6 reviews related work on validation and verification
and, finally, in section 7 we present our conclusions and point out future work.

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 (Figure 1). As
proposed in [19], 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 [12]. 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 information base, that is, insertions or deletions of instances. We assume a strict
interpretation of operation contracts [17], which prevents the application of the
operation if any constraint is violated by the state satisfying the postcondition.

The following operation contracts belong to the behavioral schema corresponding to
the structural schema in Fig. 1. Each contract describes the changes that occur in the IB
when the operation is invoked. Since we assume a strict interpretation, our contracts do
not include preconditions to guarantee the satisfaction of constraints. However, if those
preconditions were added, they would be correctly handled by our method.

The operation registerUser creates a new instance of Registered, with the
corresponding values in its attributes.
Op: registerUser(id:String, e-mail:String, c-card:String)
Pre: Registered.allInstances()->exists(u | u.oclIsNew() and u.id=id and u.e-mail=e-mail and u.credit-card=c-card)
Post: Registered.allInstances()->exists(u | u.oclIsNew() and u.id=id and u.e-mail=e-mail and u.credit-card=c-card)

The operation unregisterUser deletes the specified user from Registered and adds the user as an instance of Unregistered. This operation requires that the indicated user is registered before the execution, as specified in the precondition.

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

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 formulas in order to use a reasoning method to determine several properties on it. The subset of 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 $T_1, ..., T_n$ are terms, then $p(T_1, ..., T_n)$ or $p(T)$ is an atom. An ordinary literal is either an atom or a negated atom, i.e. $\neg p(T)$. A built-in literal has the form of $A_1 \theta A_2$, where $A_1$ and $A_2$ are terms. Operator $\theta$ is either $<$, $\leq$, $\geq$, $=$ or $\neq$.

A normal clause has the form: $A \leftarrow L_1 \land ... \land L_m$ 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 $L_1 \land \ldots \land L_m$ is the body of the clause. A fact is a normal clause of the form $p(a)$, where $p(a)$ is a ground atom. A deductive rule is a normal clause of the form: $p(T) \leftarrow L_1 \land ... \land L_m$ with $m \geq 1$, where $p$ is the derived predicate defined by the deductive rule. A condition is a formula of the (denial) form: $\leftarrow L_1 \land \ldots \land L_m$ 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 a 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 in [16] where we proposed to formalize classes, attributes and associations as base predicates, but 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 \( T_2 \) before \( T \), and has not been deleted by any operation between \( T_2 \) and \( T \). Formally, the general derivation rule is:

\[
p(P_1, P_2, \ldots, P_n, T) \leftarrow addP(P_1, P_2, \ldots, P_n, T_2) \land \neg deletedP(P_1, P_2, T_2, T) \land T_2 \leq T
\]

\[
deleP(P_1, P_2, T_1, T_2) \leftarrow deleP(P_1, P_2, T, T_1, T_2) \land T > T_1 \land T \leq T_2
\]

where \( P \) is the OID (object identifier), which is included if \( p \) is a class. \( P_1, \ldots, P_n \) 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, \( P=P_1=P \). Predicates \( addP \) and \( deleP \) 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 \( op-addP \) be an operation of the behavioral schema, with parameters \( Par_1, \ldots, Par_n \) and precondition \( pre \) such that its postcondition specifies the creation of an instance of a derived predicate \( p \). For each such operation we define the following rule:

\[
addP(P_1 Par_1, \ldots, Par_n T) \leftarrow op-addP(P_1 Par_1, \ldots, Par_n T) \land pre(T_{pre}) \land T_{pre}=T-1
\]

where \( Par_1, \ldots, Par_n \) 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 \( pre(T_{pre}) \) 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 \( op-delP \) with precondition \( pre \), that deletes an instance of \( p \) we define the derivation rule:

\[
delP(Par_1, \ldots, Par_n T) \leftarrow op-delP(Par_1, \ldots, Par_n T) \land pre(T_{pre}) \land T_{pre}=T-1
\]
where $\text{Par}_1,\ldots,\text{Par}_r$ are those parameters of the operation that identify the instance to be deleted. Thus, if $p$ is a class or association class, $\text{delP}$ 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 of a class or association class $c(C; P_1,\ldots,P_n; T)$ is added by an operation if its postcondition includes the OCL expression: $\text{C.allInstances()}\rightarrow\exists c | c.\text{oclIsNew()} \land c.\text{prop}_i = p_i$, or: $c.\text{oclIsTypeOf(C)} 
\land c.\text{prop}_i = p_i$, where each $prop_i$ is a property of $C$, which can be either an attribute or a single-valued association end.

**R2.** An instance of a binary association $r(C_1; C_2; T)$ between objects $C_1$ and $C_2$, with roles $\text{role}_1$ and $\text{role}_2$ in $r$, both with a cardinality different from 1, is added by an operation if its postcondition contains the OCL expression $c_i.\text{role}_i\rightarrow\text{includes}(c_j)$ or vice-versa. Creation or deletion of instances of n-ary associations with $n>2$ cannot be expressed in OCL if they are not association classes. The treatment of association classes is included in the previous rule, which also includes the creation of instances of binary associations where some cardinality is 1, since they are represented as terms of the corresponding class.

**R3.** An instance of a class (or association class) $c(C; P_1,\ldots,P_n; T)$ is deleted by an operation if its postcondition includes $\neg c.\text{oclIsTypeOf(C)}$ or $\text{C.allInstances()}\rightarrow\text{excludes}(c)$.

**R4.** An instance of a binary association $r(C_1; C_2; T)$ between objects $C_1$ and $C_2$, with roles $\text{role}_1$ and $\text{role}_2$ in $r$, both of them with cardinality different from 1, is deleted by an operation if its postcondition includes the OCL expression: $c_i.\text{role}_i\rightarrow\text{excludes}(c_j)$ or vice-versa.

For instance, according to the previous translation rules, the class $\text{Registered}$ of our example will be represented by means of the clauses:

\[
\text{registered}(U,Id,Email,CreditCard,T) \leftarrow \text{addRegistered}(U,Id,Email,CreditCard,T2) \wedge \neg\text{deletedRegistered}(U,T2,T) \wedge T2 \leq T
\]

\[
\text{deletedRegistered}(U,T1,T2) \leftarrow \text{delRegistered}(U,T) \wedge T > T1 \wedge T \leq T2
\]

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

\[ \text{addRegistered}(U,Id,Email,Cc,T) \leftarrow \text{registerUser}(U,Id,Email,Cc,T) \]

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 from Registered according to R3, since it includes the OCL expression \text{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 is:

\[ \text{delRegistered}(U,T) \leftarrow \text{unregisterUser}(U,T) \land \text{registered}(U,Id,E,Cc,T_{\text{pre}}) \land T_{\text{pre}}=T-1 \]

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 to be Generated

Since events cannot happen simultaneously, we need to define constraints to guarantee that two operations cannot occur at the same time. Therefore, for each operation \( o \) with parameters \( P_1,...,P_n \), we define the following constraint for each parameter \( P_i \):

\[ \leftarrow o(P_1,...,P_{i-1},1,T) \land o(P_1,...,P_{i-1},2,T) \land P_i <> P_i \]

And for each pair \( o_1, o_2 \) of operations we define the constraint:

\[ \leftarrow o_1(P_1,...,P_n,T) \land o_2(Q_1,...,Q_m,T) \]

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

\[ \leftarrow \text{unregisterUser}(U,R,T) \land \text{unregisterUser}(U2,R2,T) \land U <> U2 \]

\[ \leftarrow \text{unregisterUser}(U,R,T) \land \text{unregisterUser}(U2,R2,T) \land R <> R2 \]

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

\[ \leftarrow \text{unregisterUser}(U,R,T) \land \text{registerUser}(Id,Email,Cc,T) \]

Moreover, all constraints of the UML structural schema are translated into formulas in denial form, which represent conditions that may not hold in any state of the IB. This set of formulas is 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 validation tests that allow him to assess whether the conceptual schema being defined properly satisfies the requirements. All of them take into account both the structural and the behavioral parts of the conceptual schema.
We consider well known reasoning tasks (such as satisfiability) and predefined tests regarding operations and show how we can check them in UML schemas including operations. Finally, we also provide the designer with the ability to define his own tests to see how the schema behaves in a particular situation, and then compare the results obtained with the ones expected.

We express all these tests in terms of checking the satisfiability of a derived predicate. In this way, 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 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. Checking Strong Satisfiability

A schema is strongly satisfiable if there is at least one fully populated state of the IB satisfying all the constraints [13]. 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 if there exists an instance of all classes and associations of the schema. In our example:

\[
\text{sat} \leftarrow \text{registered}(U, \text{Uid}, \text{Email}, \text{C-card}, T) \land \text{unregistered}(U, \text{Uid2}, \text{Email2}, \text{Reason}, T) \\
\land \text{product}(P, \text{Pid}, \text{Descr}, \text{St-price}, \text{Owner}, T) \land \text{bid}(B, \text{Product}, \text{Bidder}, \text{Amount}, T)
\]

It can easily be seen that the schema of our example is not strongly satisfiable, since the operations of the behavioral schema do not allow creating an instance of Registered owning at least one Product. To avoid this mistake, we replace the original operation registerUser by the following one which is responsible for creating both an instance of Registered and the corresponding instance of Product that will be offered by him:

**Op:** registerUser(id: String, email: String, cc: String, pid: String, descr: String, st-price: Float)

**Pre:**

**Post:**

\[-\text{create a registered user } u\\n\text{Registered.allInstances() \to exists(u | u.oclIsNew())}\\na n d u.\text{e-mail} = \text{email} \text{ and } u.\text{c-card} = \text{cc} \text{ and }\\n\text{product.allInstances() \to exists(p | p.oclIsNew() and p.id=pid}\\na n d p.\text{desc}r = \text{descr} \text{ and } p.\text{starting-price} = \text{st-price} \text{ and }\\n\text{p.\text{owner.e-mail} = email})
\]

Also, we need an additional operation to create instances of Bid. We define the following contract for placeBid stating that an instance of bid, associated to the indicated user and product is created. Its precondition states that bidders must be registered and must own some product to be able to bid:

**Op:** placeBid(bidder:User, prod:Product, amount:Float)

**Pre:**

**Post:**

\[-\text{bidder.oclIsTypeOf(Registered) and bidder.product \to notEmpty()}
\land \text{Bid.allInstances() \to exists(b | b.oclIsNew() and b.bidder = bidder and b.product = product}\\na n d b.\text{bidder} = \text{bidder} \text{ and } b.\text{product} = \text{product} \text{ and } b.\text{amount} = \text{amount})
\]
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 5. 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 certain state that is valid according to the schema:

```java
registerUser(john, john@upc.edu, 111, p1, pen, 10, 1), unregisterUser(john, 2),
registerUser(mary, mary@upc.edu, 222, p2, pen, 20, 3),
registerUser(peter, peter@upc.edu, 333, p3, pen, 30, 4), placeBid(peter, p2, 25, 5)
```

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 registered user Mary that offers a pen to have an instance of Registered. Finally, to populate the class Bid we need to register another user Peter that bids for the pen offered by Mary.

### 4.2. Testing Properties of the Operations

When dealing with operations, additional validation tests can be performed, namely applicability and executability of each operation [5]. To illustrate these properties, let us consider an additional operation removeUser, that deletes the specified user as long as he or she is not the owner of any product and has not hidden for any product:

- **Op:** removeUser(u: User)
- **Pre:** u.offered-prod->isEmpty() and u.bid->isEmpty()
- **Post:** User.allInstances()->excludes(u)

As can be seen, the precondition of this operation requires the existence of at least a user not offering any product, which is not possible according to the cardinality constraint 1..* of offered-prod. This means that this operation is not applicable, and the designer should avoid this situation by, for example, removing the first part of the precondition. The formalization for an operation $O$ with precondition $\text{pre}(t)$ is:

```
\text{applicable}_O \leftarrow \text{pre}(T)
```

If $\text{applicable}_O$ is not satisfiable, the operation $O$ 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 again the operation removeUser, and assume now that its first precondition has been eliminated. Now the 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 of owner. 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 schema to record the execution of the operation. Now, if executed, $O$ is satisfiable, then $O$ is executable:

```
\text{executed}_O \leftarrow o(P_1,...,P_n T) \land \text{pre}(T_{pre}) \land T_{pre}=T-1
```
4.3. User-defined Validation Tests

Although there are some tests that can be performed without the intervention of the designer as the previous ones, the validation process is not fully formalizable. The reason is that validating a conceptual schema consists in checking if it correctly specifies the requirements, which are not usually formalized. This means that it is desirable to help the designer to analyze the schema so that he can decide whether it represents the intended domain.

To do this, during the validation process, the designer will need to define his own derived predicates, as the generic ones we have explained in the previous subsections. In this way, he will be able to determine whether the schema is correct by executing these user-defined tests and comparing the obtained results to those expected according to the requirements. For instance, an interesting question could be “Can the system store bids of unregistered users?”. The designer will have to formalize the derivation rule:

\[
bidsOfUnreg \leftarrow bid(B,Prod,Usr,Amt,T) \land \neg \text{registered}(Usr,Id,Email,Reason,T)
\]

In this case, \(bidsOfUnreg\) is satisfiable, as shown by the sample instantiation:

\{registerUser(john, john@upc.edu, 111, prod1, pen, 10, 1),
registerUser(mary, mary@upc.edu, 222, prod2, pen, 20, 2),
placeBid(mary, prod1, 15, 3), unregisterUser(mary, 4)\}

By studying the results of the 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.

5. Experimental Evaluation

We have studied the feasibility of our approach by using an implementation of an existing reasoning procedure, called CQC-Method [8]. 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.

The CQC Method performs query containment tests on deductive database schemas. It is a semidecidable procedure for finite satisfiability and unsatisfiability. This means that it always terminates when there exists a finite consistent state satisfying the property, or when the property is unsatisfiable (finitely or infinitely).

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 database schema definition itself. The goal to attain, which must be achieved on the database 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)** according to the syntactic properties of the conceptual schema considered in each test. The method maintains an account of the constants that appear in the initial goal and in the definition of the schema, or that have been introduced in previous instantiations.

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 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 values are 2 to 5. It may happen that the goal succeeds or fails without further instantiations, and then \( d \) 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, when attempting to derive an **Unregistered** user which must hold at time \( t \), our set of temporal constants may be \( \{1, d, 5\} \), being \( d \) a virtual constant. 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 \( (T \emptyset KT_t) \) to \( (\emptyset \theta KT_{t+1}) \) that instantiates the temporal variable \( T \) according to one of the VIP-rules, where \( \theta \) is a ground substitution of \( T \) and \( KT_t \) 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 \( G_i \) 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 \( T = k_{i-1}, k_i \in KT_i \):

- **Tmp1.** \( \theta = T/c \) and \( KT_{t+1} = KT_t \), where \( c = c_{i-1}, T \subseteq KT_t \) \( \{T = c_{i-1}\} \in G_i \)
6. Related Work

The problem most commonly addressed in the context of validating ER schemas has been the satisfiability of cardinality constraints [7, 11, 13]. In UML schemas, a well-known approach is to translate them into Description Logics (DL) and then use current standard DL-based reasoning systems to automatically verify properties like satisfiability, class equivalence or class subsumption [2]. An approach considering general-purpose constraints is [16], in which the schema is translated into logic in order to check properties such as schema satisfiability, satisfiability of classes and associations or constraint redundancy. The main limitation regarding all these approaches is that none of them takes into account the behavioral schema in the determination of the correctness of the structural schema.

One of the first methods that deals with the behavioral schema is in the context of deductive databases [6]. To validate the schema, the designer asks how a given state can be reached, but integrity constraints are not taken into account. Moreover, some structural features, such as inheritance and derivation rules are not supported. In the same context, [5] identifies a set of interesting properties on a conceptual schema taking into account its behavioral part, and proposes a framework to check these properties on a deductive conceptual schema by means of planning.

A few proposals address verification and validation in object-oriented conceptual schemas with a behavioral part. In [9] an approach is proposed to check the consistency between integrity constraints and the transitions of statechart diagrams. However, it does not consider general operations and the constraints and the postconditions of a transition may only express comparison conditions between an attribute and a constant.

An approach to reason on UML/OCL schemas is HOL-OCL [3]. The method uses a theorem prover to determine some properties on the schema, such as equivalence of two integrity constraints. The theorems to be proved are defined in terms of the metamodel and, thus, it is not possible to check whether a certain instantiation is accepted by a schema or which is the sequence of operations that leads to a certain state.

An interesting tool to validate UML/OCL conceptual schemas is USE [10], 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, inserting and deleting instances of the model, and then asking the tool to test whether the instantiation
satisfies the postcondition. Since the instantiations must be manually provided, this tool has some drawbacks. For instance, it cannot automatically verify that the definition of the schema satisfies certain properties. Moreover, it cannot validate that the schema accepts an information base containing a subset of information defined declaratively.

We may note that all object-oriented approaches that consider the behavioral part may report as valid a state satisfying all the constraints but that is impossible to construct using the operations defined in the schema. Moreover, they cannot automatically construct the sequence of operations resulting in a certain state. The main reason for this weakness is that they do not take into account the definition of the operations when determining if a state is accepted or not by the schema.

An exception is an approach that belongs to the Rodin project. It combines UML-B [18] and ProB [14], the former to represent the schema and translate it into the B language, and the latter to validate it by animation. One of its drawbacks is that UML-B only accepts a subset of the UML that is suitable for translation into B, which is defined through an ad-hoc profile. Moreover, constraints and operations must be directly expressed in B by the designer. In contrast, our models can be expressed in standard UML and OCL, which are the languages most commonly used in conceptual modeling. Regarding the animation process, the operations handled by ProB must incorporate the semantics of the constraints, which are checked after the simulated execution. On the contrary, our approach is able to deal with constraints as such, taking care of maintaining them while constructing the sample state. Additionally, ProB requires that the state 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 by this approach.

7. Conclusions and Further Work

We have proposed a new approach to validate a complete UML conceptual schema, with textual OCL integrity constraints and OCL pre and postconditions of operation contracts. Our approach allows automatically determining 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 operation executions 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 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 and applying it to our example.

There are some interesting directions for further work. First, we plan to provide an implementation of the translation of the UML conceptual schema into logic. Also, we
plan to extend our approach to be able to validate conceptual schemas in the presence of derived UML information such as attributes, classes or associations.

Acknowledgements. This work has been partly supported by the Ministerio de Ciencia y Tecnología under projects TIN2005-06053 and TIN2005-05406.

References