Validation of UML Conceptual Schemas with Operations

Anna Queralt and Ernest Teniente

Universitat Politècnica de Catalunya
Dept. de Llenguatges i Sistemes Informàtics
c/ Jordi Girona 1-3, 08034 Barcelona (Catalonia, Spain)
{aqueralt, teniente}@lsi.upc.edu

Abstract. The purpose of validating a conceptual schema is to guarantee that it properly reflects what the user needs from an application. 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. A conceptual schema specifies the relevant information about the domain, and how this information changes as a result of operations. In this sense, we propose an approach to validate a UML conceptual schema by simulating the execution of the operations defined in it, so that the designer can check both that the schema is correctly defined and that it satisfies the requirements.

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 validation corresponds to the question *Am I building the right system?*. While it is true that software quality can be understood in terms of usability, reliability or efficiency, and these properties can mostly be 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. Moreover, 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. In fact, this has been identified as one of the key problems to be solved for achieving the goal of automating information systems building [14].

Validation can be used to assure the quality of a conceptual schema instead of a piece of code. In this case, validation aims to guarantee that a specification properly reflects what the user needs from an application. The validation task is not fully formalizable and it is based on intuition. Then, it is desirable to provide the designer with a set of tools that assist him or her in the validation process [4]. Rather than using informal techniques, such as building a prototype which shows the behavior of the application, a better option is to animate the specification itself.

Animation is one of the most common approaches to validation. With animation techniques, the schema, which must be specified using a formal language, is executed
without translating it into an implementation [5]. Basically, animation consists in the population of the schema and then examining the results, which are then checked by the designer against the requirements.

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.

In our previous work [15] we proposed an approach to determine the correctness of the structural part of a conceptual schema by defining a set of properties that it must fulfill, such as schema satisfiability, checking them by means of the CQC Method [9]. However, although the correctness of the structural schema is a premise for the correctness of the complete conceptual schema, it does not guarantee that the behavioral part appropriately manages the information according to the requirements.

In fact, it may perfectly happen that the states of the information base which show that the structural schema satisfies a certain desirable property are never reachable by means of the operations defined in the behavioral part. Thus, a complete validation of the conceptual schema cannot be performed without taking into account both the structural and the behavioral parts.

Also, new desirable properties appear when dealing with the behavioral schema, namely applicability and executability of operations, and reasoning tasks regarding the structural schema need to be adapted. For instance, when dealing with the structural schema alone, satisfiability means the existence of an instantiation fulfilling all the integrity constraints. Now, the schema will be satisfiable if there exists a correct sequence of operations that leads to an instantiation fulfilling all the constraints.

In this paper we propose an approach to validate a complete conceptual schema (structural and behavioral) by means of animation. The structural part of the schema is formalized in UML and OCL, and the behavioral part consists of a set of operations formalized in OCL. Then, the execution of operations is simulated with the aim to determine if the schema behaves as expected according to the requirements. Two different kinds of reasoning are provided: determining whether the conceptual schema is correctly defined and whether, being correct, satisfies the user requirements.

Our approach consists in using the CQC Method to populate the schema, taking into account that the only changes accepted are those defined in the operations of the behavioral schema. To do this, we have had to translate the UML/OCL conceptual schema into logic in such a way that the logic representation incorporates the effect of operation execution in terms of the instances of classes and associations that exist at a certain time. Moreover, we have had to extend the CQC Method so that it can properly deal with the temporal features of the operations.

The different meaning of reasoning tasks when dealing with operations is discussed in section 2, where we also introduce a running example. Section 3 presents our approach, both the translation of the complete schema into logic in order to use the CQC Method, and the extension of this method to deal with operations. In section 4 we illustrate the validation of a conceptual schema using the CQC Method in our example. Section 5 reviews related work on validation and verification of conceptual schemas and, finally, in section 6 we present our conclusions and point out future work.
2. Validating with and without Operations

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 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. As proposed in [18], we will assume these constraints are specified in OCL.

Fig. 1 shows the structural schema of a (simplified) on-line auction site. The system stores information about users. Some of them are registered, and then have a credit card, and the rest are 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.

![Fig. 1. The structural schema of an on-line auction site](image)

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 they define the only changes that can be performed on the IB.

An operation is defined by means of 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.

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. According to the strict interpretation of operation contracts [16], preconditions need not be responsible for guaranteeing the satisfaction of integrity constraints. In this sense, it is assumed that constraints are checked at the end of each execution and the operation is rejected in case some constraint is violated.

**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
This is why possible violations of integrity constraints are not prevented in the preconditions of our contracts.

The operation `registerUser` creates a new instance of `Registered`, with the corresponding values in its attributes.

\[\text{Op: registerUser(id: String, e-mail: String, c-card: String)}\]
\[\text{Pre: } u.oclIsNew() \text{ and } u.oclIsTypeOf(Registered) \text{ and } u.id = id\]
\[\text{and } u.e-mail = e-mail \text{ and } u.credit-card = c-card\]

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.

\[\text{Op: unregisterUser(u: User, reason: String)}\]
\[\text{Pre: } u.oclIsTypeOf(Registered)\]
\[\text{Post: } u.oclIsTypeOF(Unregistered) \text{ and } u.reason = reason\]
\[\text{and not } u.oclIsTypeOf(Registered)\]

Reasoning on the structural schema of Fig. 1 alone, it may be easily determined that it does not present any of the misspecifications that have been usually reported in the literature, such as unsatisfiability of the schema, unliveness of a class or association or redundancy of a constraint. Thus, we can see that its definition is semantically correct. For instance, the following state of the IB:

"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 correct 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.

In our example, although it is possible to find instances of `Registered` 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, which violates the cardinality constraint of the role `offered>prod`. Thus, although the structural part of the conceptual schema is semantically correct, the complete conceptual schema is not.

Also, when considering the behavioral schema, new kinds of reasoning tasks appear that must be taken into account to check the correctness of the whole conceptual schema. For instance, there may exist operations that can never be applied because their precondition cannot be satisfied. It might also occur that an operation, despite being applicable, may never be successfully executed because it always leaves the IB in an inconsistent state (as happens in our example with `registerUser`). These are also undesirable situations that the designer should avoid by modifying either the structural schema or the behavioral one.

In section 4 we exemplify these and further flaws that may be found in a conceptual schema.
3. Our Approach to Validation

The approach we propose to validation consists in using the CQC Method to animate the conceptual schema and determine if it satisfies a set of properties. To do this, the structural and behavioral parts of the conceptual schema need to be translated into a set of logic formulas that can be interpreted by the CQC Method. Moreover, we must extend the CQC Method so that it can properly deal with the operations.

We explain first how the complete schema is translated into logic and then present the CQC Method and the extensions needed in order to deal with the operations.

3.1. Translation of a UML Conceptual Schema into Logic

Reasoning tasks which consider the structural schema alone are aimed at checking that an instantiation fulfilling a certain property and satisfying the integrity constraints can exist. For this purpose, classes, attributes and associations can be translated into basic 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.

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 (assuming they are correct), 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 [15] where we proposed to formalize classes, attributes and associations as base predicates. As we have just seen, this formalization does not suffice to ensure that instances can only exist as a result of an event.

Translation of the Structural Schema

Classes and associations are represented by means of derived predicates whose derivation rules ensure that their instances are given by the occurrence of operations. Then, an instance of a derived predicate $P$ 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 for a class or an association is:

$$P(p_1,\ldots,p_n,t) \leftarrow addP(p_1,\ldots,p_n,t_2) \land \neg deletedP(p_1,\ldots,p_n,t_2, t) \land t_2 \leq t$$

$$deletedP(p_1,\ldots,p_n,t_1,t_2) \leftarrow delP(p_1,\ldots,p_n,t) \land t_1 > t \land t_1 \leq t_2$$
where \( p_i, \ldots, p_j \) are the terms of \( P \) that suffice to identify an instance of \( P \). Predicates \( \text{add}P \) and \( \text{del}P \) are also derived predicates that hold if some operation has created or deleted an instance of \( P \) at time \( t \), respectively. In the next subsection we will see how their derivation rules are obtained, since they depend on the operations of the behavioral schema.

To simplify the definition of these derivation rules, we have modified the translation of our previous proposal [15]. In particular, instead of defining separate predicates for each attribute and association end of a class, we add them as terms of the predicate representing the class, as long as their cardinality is 1, since the life of their instances coincides with that of the instances of the class to which they belong. Additionally, we include the inherited information in the subclasses, and define the superclasses as derived from the instances of their subclasses. We also need to add a term \( t \), representing the time in which an instance exists.

For instance, according to this translation the hierarchy of users of our example will be represented by means of the derived predicates:

\[
\begin{align*}
\text{Registered}(u, \text{id}, \text{e-mail}, \text{credit-card}, t) \\
\text{Unregistered}(u, \text{id}, \text{e-mail}, \text{reason}, t)
\end{align*}
\]

\[
\text{User}(u, \text{id}, \text{e-mail}, t) \leftarrow \text{Registered}(u, \text{id}, \text{e-mail}, \text{credit-card}, t)
\]

\[
\text{User}(u, \text{id}, \text{e-mail}, t) \leftarrow \text{Unregistered}(u, \text{id}, \text{e-mail}, \text{reason}, t)
\]

where \( u \) corresponds to the unique object identifier (OID) required by every instance of a class, and the derivation rules for \( \text{Registered} \) and \( \text{Unregistered} \) must be specified according to the general rule above. For instance, the one for \( \text{Registered} \) is:

\[
\text{Registered}(u, \text{id}, \text{e-mail}, \text{credit-card}, t) \leftarrow \text{addRegistered}(u, \text{id}, \text{e-mail}, \text{credit-card}, t_2) \land \neg \text{deletedRegistered}(u, t_2, t) \land t_2 \leq t
\]

\[
\text{deletedRegistered}(u, t_1, t_2) \leftarrow \text{delRegistered}(u, t) \land t > t_1 \land t \leq t_2
\]

In turn, \( \text{addRegistered} \) and \( \text{delRegistered} \) are derived predicates whose definition depends on the operations of the behavioral schema that may insert and delete instances of the class \( \text{Registered} \) as we will see in the next subsection.

In addition to classes, attributes and associations, a UML class diagram includes a set of constraints. These constraints can be implicit, such as referential constraints in associations or uniqueness of OIDs, or explicit, such as cardinalities or textual constraints. All of them are translated into formulas in denial form, which represent conditions that must not be satisfied in any state of the IB. The set of constraints to be generated is the one resulting from the translation of the structural schema [15], but taking into account the new representation of classes. Now these constraints are defined in terms of derived predicates, since all the classes and associations are derived. This is not a problem, since the CQC Method, which we are going to use to animate the schema, is able to manage this situation.

For instance, the implicit referential constraint of the association \( \text{Offered by} \) stating that the \textit{owner} of a \textit{Product} must be a \textit{User} is formalized as follows:

\[
\leftarrow \text{Product}(p, \text{id}, \text{descr}, \text{st-price}, \text{owner}, t) \land \neg \text{User}(\text{owner}, \text{id}, \text{e-mail}, t)
\]

This formalization guarantees at the same time the cardinality constraint 1 of \textit{User} in association \textit{Offered by}, since the predicate \textit{Product} includes a term \textit{owner} that refers to
The formalization of the explicit cardinality constraint 1..* of Product in the same association is:

\[ \leftarrow \text{User}(u, id, e-mail, t) \land \neg \text{Product}(p, id, descr, st-price, u, t) \]

And the textual OCL constraint stating that users are identified by id is as follows:

\[ \leftarrow \text{User}(u, id, e-mail, t) \land \text{User}(u2, id2, e-mail2, t) \land u <> u2 \land id = id2 \]

**Translation of the Behavioral Schema**

As we said, the operations of the behavioral schema are translated into base predicates in our logic representation. In addition to the parameters of the operation, base predicates require an additional term \( t \) representing the occurrence time of the operation to be able to determine which instances of a derived predicate exist at \( t \). This term will also indicate the order of execution of each operation within a sequence leading to a certain state of the IB.

Thus, for each operation \( O \) with parameters \( p_1, ..., p_n \) we define a base predicate \( O \), with a term for each parameter and an additional term \( t \) that represents the occurrence time of the event. For instance, the operation unregisterUser of our example is translated into the base predicate unregisterUser\((u, reason, t)\).

Additionally, since events cannot happen simultaneously, we need to define constraints that guarantee that two operations cannot occur at the same time. For each operation \( O \) with parameters \( p_1, ..., p_n \) we define the following constraint for each parameter \( p_i \):

\[ \leftarrow O(p_1, ..., p_n, t) \land O(p_1', ..., p_n', t) \land p_i <> p_i' \]

And for each pair \( O, O2 \) of operations we define the constraint:

\[ \leftarrow O(p_1, ..., p_n, t) \land O2(q_1, ..., q_m, t) \]

In our example, the predicate unregisterUser\((u, reason, t)\) requires the constraints:

\[ \leftarrow \text{unregisterUser}(u, r, t) \land \text{unregisterUser}(u2, r2, t) \land u <> u2 \]
\[ \leftarrow \neg \text{unregisterUser}(u, r, t) \land \text{registerUser}(id, email, cc, t) \]

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

\[ \leftarrow \text{unregisterUser}(u, r, t) \land \text{registerUser}(id, email, cc, t) \]

In order to obtain the complete translation of the schema, the derivation rules of the predicates \( addP \) and \( deleteP \) have to be drawn from the operations.

Let \( Op\rightarrow addP \) be an operation of the behavioral schema, with parameters \( p_1, ..., p_n \) 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:

\[ addP([p], p_0, ..., p_n, t) \leftarrow Op\rightarrow addP(p_0, ..., p_n, t) \land Pre(t_{pre}) \land t_{pre}=t-1 \]  
[ \land \text{oid}(p) ]

where \( p_0, ..., p_n \) 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. In case \( P \) is a class, an OID, which is not a parameter of the operation, is required. This is represented by the base predicate \( \text{oid}(p) \), that must be added to the rule in this case, as well as the term \( p \) in \( addP \). \( Pre(t_{pre}) \) is the translation of the precondition of the operation, following the same rules used to translate OCL integrity constraints. Note that, since the precondition must hold before the occurrence of the operation, the time of all the facts appearing in it must be \( t-1 \).
Similarly, for each operation $Op_{delP}(p_1, \ldots p_n, t)$ with precondition $Pre_i$, that deletes an instance of $P$ we define the derivation rule:

$$delP(p_\delta, \ldots p_n, t) \leftarrow Op_{delP}(p_1, \ldots p_n, t) \wedge Pre_i(t_{pre}) \wedge t_{pre}=t-1$$

where $p_\delta, \ldots p_n$ are those parameters of the operation that identify the instance to be deleted. Thus, if $P$ is a class, $delP$ will have a single term in addition to $t$, which corresponds to the OID of the deleted instance.

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

- An instance of a predicate $C(c,p_1, \ldots p_n, t)$ representing a class (or association class) is added by an operation if its postcondition includes the OCL expression: $c.oclIsTypeOf(C)$ and $c.prop_1=p_1$ and ... $c.prop_n=p_n$, where each $prop_i$ is a property of the class, which can be either an attribute or an association end with exactly one object.

- An instance of a predicate $R(c_1, c_2, t)$ representing a binary association between objects $c_1$ and $c_2$, with roles $role_1$ and $role_2$ in $R$, both of them with cardinality different from 1, is added by an operation if its postcondition contains the OCL expression $c_1.role-> includes(c_2)$ 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.

- An instance of a predicate $C(c,p_1, \ldots p_n, t)$ representing a class (or association class) $C$ is deleted by an operation if its postcondition includes the OCL expression: $not c.oclIsTypeOf(C)$.

- An instance of a predicate $R(c_1, c_2, t)$ representing a binary association between objects $c_1$ and $c_2$, with roles $role_1$ and $role_2$ in $R$, both of them with cardinality different from 1, is deleted by an operation if its postcondition includes the OCL expression: $c_1.role-> excludes(c_2)$ or vice-versa.

We will apply these rules to our example to provide the derivation rules for $addRegistered(u, id, e-mail, credit-card, t)$ and $delRegistered(u, t)$ which are required to complete the definition of our predicate $Registered$.

The operation $registerUser$ creates an instance of $Registered(u, id, e-mail, credit-card, t)$, since its postcondition includes the OCL expression $u.oclIsTypeOf(Registered)$.
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:

\[ \text{addRegistered}(u, id, e-mail, cc, t) \leftarrow \text{registerUser}(id, e-mail, cc, t) \wedge \text{oid}(u) \]

Now we need to find which operations are responsible for deleting instances of Registered in order to specify the derivation rule of delRegistered. We can see that the operation unregisterUser is the only one that 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 is: in this case will be:

\[ \text{delRegistered}(u, t) \leftarrow \text{unregisterUser}(u, t) \wedge \text{Registered}(u, id, e-mail, cc, t_{\text{pre}}) \wedge t_{\text{pre}}=t-1 \]

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. The CQC Method

The CQC Method performs query containment tests on deductive database schemas, that can also be used to determine several properties on a database schema [9]. 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. As we will see in the next section, 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 Engine to attempt to construct a sample IB to prove that the initial goal \(G_0\) is satisfied without violating any integrity constraint in \(F_0\). When applied to our logical formalization of a conceptual schema, the sample IBs obtained by the method are defined by means of a sequence of operation calls.

#### 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 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, 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 \((T \not\emptyset KT)\) to \((\emptyset \not\emptyset KT_{t+1})\) that instantiates the temporal variable T according to one of the VIP-rules, where \(\theta\) is a ground substitution of T and KT, 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 following VIP-rules.Tmp1 to Tmp4 apply when instantiating a temporal variable T such that \(T = k_{i-1}, k_i \in KT\), while Tmp5 to Tmp10 are adapted from the discrete order VIP and are applied when instantiating a temporal variable T that does not satisfy the previous condition.

\[
\text{Tmp1. } \theta = T/c_{prev} \text{ and } KT_{t+1} = KT, \text{ where } c_{prev} = c_{\text{ suc}-1}, \{c_{\text{ suc}}, c_{prev}\} \subseteq KT, \{T = c_{\text{ suc}-1}\} \in G_c
\]

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

\[
\text{Tmp3. } \theta = T/c_{\text{ new}} \text{ and } KT_{t+1} = KT \cup \{c_{\text{ new}}\}, \text{ where } c_{\text{ new}} = c_{\text{ suc}-1}, c_{\text{ new}} \in KT, c_{\text{ suc}} \in KT, \{T = c_{\text{ suc}-1}\} \in G_c, \text{ there is no } d_{prev} \text{ tied to } c_{\text{ suc}} \text{ in } KT, \text{ and there is no } c_{prev} \in KT \text{ such that } c_{prev} < c_{\text{ suc}} \text{ and } |\{d, |d| \in KT, c_{\text{ suc}} < d < c_{\text{ suc}}| < |c_{\text{ suc}} - c_{prev}| - 1.
\]

\[
\text{Tmp4. } \theta = T/d_{\text{ new}} \text{ and } KT_{t+1} = KT \cup \{d_{\text{ new}}\}, \text{ where } d_{\text{ new}} \in KT, d_{\text{ suc}} \in KT, \{T = d_{\text{ suc}-1}\} \in G_c, \text{ there is no } d_{prev} \text{ tied to } d_{\text{ suc}} \text{ in } KT, \text{ and } d_{\text{ new}} \text{ is tied to } d_{\text{ suc}} \text{ in } KT_{t+1} \text{ and}
\]
there are no \( c_p, c_j \in KT \) such that \( c_i < d_{\text{new}} < c_j \) and there is no \( c_m \) with \( c_i < c_m < c_j \) and \( |\{d_i \mid d_i \in KT, c_i < d_i < c_j\}| < |c_j - c_i| - 1 \).

\[
\text{Tmp5. } \theta = T/k \text{ and } KT_{i+1} = KT, \text{ where } k \in KT,
\]

\[
\text{Tmp6. } \theta = T/d_{\text{new}} \text{ and } KT_{i+1} = KT \cup \{d_{\text{new}}\}, \text{ where } d_{\text{new}} < \min(KT)
\]

\[
\text{Tmp7. } \theta = T/d_{\text{new}} \text{ and } KT_{i+1} = KT \cup \{d_{\text{new}}\}, \text{ where } \min(KT) \leq d < d_{\text{new}} < k_{i+1} \leq c_{\text{min}}, \text{ there is no } d \in KT \text{ such that } d < k_{i+1} \text{ and there is no } c_p \in KT \text{ such that } c_p < c_{\text{min}}
\]

\[
\text{Tmp8. } \theta = T/d_{\text{new}} \text{ and } KT_{i+1} = KT \cup \{d_{\text{new}}\}, \text{ where } c_{\text{max}} \leq k_j < d_{\text{new}} < k_{j+1} \leq c_{j+1}, \text{ there is no } d \in KT \text{ such that } k_j < d < k_{j+1} \text{ and there is no } c_p \in KT \text{ such that } c_j < c_p < c_{j+1} \text{ and } |\{d_i \mid d_i \in KT, c_i < d_i < c_j\}| < |c_j - c_i| - 1
\]

\[
\text{Tmp9. } \theta = T/d_{\text{new}} \text{ and } KT_{i+1} = KT \cup \{d_{\text{new}}\}, \text{ where } c_{\text{max}} \leq k_j < d_{\text{new}} < d_{j+1} \leq \max(KT), \text{ there is no } d \in KT \text{ such that } k_j < d < d_{j+1} \text{ and there is no } c_p \in KT \text{ such that } c_{\text{max}} < c_p
\]

\[
\text{Tmp10. } \theta = T/d_{\text{new}} \text{ and } KT_{i+1} = KT \cup \{d_{\text{new}}\}, \text{ where } \max(KT) < d_{\text{new}}
\]

4. Validation of a Conceptual Schema

We will now illustrate how our approach can be used to determine the correctness of a conceptual schema from two different points of view. First, the designer must be sure that the conceptual schema is correctly defined. This is usually known as verification and can be done automatically. Second, the designer must check that the schema defined is the right conceptual schema, that is, it correctly specifies the requirements. This process corresponds to validation and must be guided by the designer. The CQC Method, with the extensions proposed in this work, is used to both ends.

4.1. Is the Conceptual Schema Right?

In [15] we proposed an approach to check a set of desirable properties of the structural schema, such as satisfiability or liveness. A schema is satisfiable if there is a non-empty state of the IB in which all integrity constraints are satisfied, and a class or association is lively if it can have at least one instance satisfying all constraints. For the purpose of validation, a more interesting property to be verified is strong satisfiability [13], which ensures that there is at least one fully populated state of the IB satisfying all constraints. In the presence of operations, this means checking whether they allow to create at least a complete instantiation satisfying all integrity constraints.

In order to use the CQC Method to check whether a certain property holds, a goal \( G_0 \) must be formulated, and the method will try to construct a sample IB fulfilling that goal and the set of constraints defined in the schema. Thus, the goal \( G_0 \) we must check to ensure that the schema is strongly satisfiable is to have an instance of all classes and associations of the schema. In our example:
\[ G_0 = \text{Registered}(u, \text{uid}, \text{email}, \text{c-card}, t) \land \text{Unregistered}(u2, \text{uid2}, \text{email2}, \text{reason}, t) \land \text{Product}(p, \text{pid}, \text{descr}, \text{st-price}, \text{owner}, t) \land \text{Bid}(b, \text{product}, \text{bidder}, \text{amount}, t) \]

Note that the goal is expressed in terms of the structural schema. This means that all the reasoning tasks proposed in our previous work can be performed in the same way when dealing with operations, although they now have a different meaning. The new translation of the schema into logic, defining the population of classes and associations as derived from operations, ensures that an instance can only exist if there is an operation responsible for creating it correctly.

As we have seen in section 2, the schema of our example is not satisfiable, since it is impossible to create an instance of \text{Registered} owning at least one \text{Product}. To avoid this mistake, we replace the original operation \text{registerUser} by the following one which is responsible for creating both an instance of \text{Registered} and the corresponding instance of \text{Product} that will be offered by the new registered user:

\[
\text{Op: registerUser}(\text{id}: \text{String}, \text{email}: \text{String}, \text{cc}: \text{String}, \text{pid}: \text{String}, \text{descr}: \text{String}, \text{st-price}: \text{Float})
\]

\[
\text{Pre:}
\]

\[
\text{Post: --create a registered user } u \\
\text{u.oclIsNew()} \land \text{u.oclIsTypeOf(Registered)} \land \text{u.e-mail= email} \land \text{u.c-card=cc} \land \\
\text{--create a Product p and associate it to u} \\
\text{p.oclIsNew()} \land \text{p.oclIsTypeOf(Product)} \land \text{p.id=pid} \land \text{p.descr=descr} \land \text{p.starting-price=st-price} \land \text{p.owner=u}
\]

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

\[
\text{Op: placeBid}(\text{bidder:User}, \text{prod:Product}, \text{amount:Float})
\]

\[
\text{Pre: bidder.oclIsTypeOf(Registered)} \land \text{bidder.product->notEmpty()} \\
\text{Post: b.oclIsNew()} \land \text{b.oclIsTypeOf(Bid)} \land \text{b.bidder = bidder} \land \text{b.product = product} \land \text{b.amount = amount}
\]

The logic representation of \text{Registered}, \text{Unregistered}, \text{Product} and \text{Bid} according to this new behavioral schema can be found in the Appendix of [17]. Now, asking the CQC Method to satisfy the above goal \( G_0 \), the answer is that the schema is strongly satisfiable as shown by the following instantiation, which implies that all classes are populated at time 5:

\{\text{registerUser(John, john@upc.edu, 111, p1, pen, 10, 1)}, \text{unregisterUser(John, 2)}, \text{registerUser(Mary, mary@upc.edu, 222, p2, pen, 20, 3)}, \text{registerUser(Peter, peter@upc.edu, 333, p3, pen, 30, 4)}, \text{placeBid(Peter, p2, 25, 5)}\}

When dealing with operations, additional reasoning tasks can be checked, namely applicability and executability [6]. 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 bid for any product:

- **Op:** `removeUser(u: User)`
- **Pre:** `u.offered-prod->isEmpty()` and `u.bid->isEmpty()`
- **Post:** `not u.oclIsTypeOf(User)`

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. In order to check applicability of an operation with the CQC Method, \( G_0 = \text{Pre} \). In case the goal fails, 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 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 translation of the schema to record the execution of the operation:

\[
\text{executed}_O \leftarrow O(p_1,\ldots,p_n,t) \land \text{Pre}(t)\]

and the goal will be \( G_0 = \text{executed}_O \). If it succeeds, \( O \) is executable.

### 4.2. Is It the Right Conceptual Schema?

Once we are sure that the schema is correctly defined, we may also help the designer in detecting potentially undesirable situations admitted by the schema. This can be done by automatically generating questions of interest that can be drawn from the definition of the schema.

For instance, in our example, although a user can be the owner of several products according to the cardinality constraint 1..* of offered-prod, he cannot own more than one product in practice with the given operations of the behavioral schema. This does not mean that the schema is not correctly defined, since an IB with a single product per user satisfies all the constraints. However, there is probably something that the designer overlooked when specifying the behavioral schema like, for instance, an operation to allow existing users to offer new products.

The goal to detect the previous situation, which can be automatically generated from the information provided by the conceptual schema, is the following:

\[
G_0 = \exists p, p_2. (\text{Product}(p,\ldots,p_n,t) \land \text{Pre}(t))
\]

Since this goal fails (i.e. the CQC Method does not find any sample instantiation that satisfies it), then a user cannot be the owner of several products, although it should be possible according to the cardinality 1..*.
We can also generate questions of interest from the behavioral schema. For instance, the second precondition of placeBid states that the bidder must be the owner of some product. According to the structural schema, this is always true due to the cardinality 1..* of offered-prod, which means that this precondition is redundant since it will never be violated. The goal that must be formulated to the CQC Method to solve this question is:

\[ G_0 = \text{User}(u, \text{id, e-mail, t}) \land \neg \text{offersProduct}(u, t) \]

\[ \text{offersProduct}(u, t) \leftarrow \text{Product}(p, \text{id, descr, st-price, u, t}) \]

In this case, failure of \( G_0 \) means that there is a potentially undesirable situation, i.e. that the precondition of placeBid is redundant since the conceptual schema guarantees that it will always be satisfied.

There are several kinds of questions of interest that can be automatically obtained from the definition of both the structural and the behavioral schema. We provide here the general form of the goals to check some of them, including those mentioned before. In the following cases, if \( G_0 \) fails means that there is a potentially undesirable situation:

- For each precondition \( Pre \) of each operation, is it redundant?
  \[ G_0 = \neg Pre \]
- For each cardinality constraint 1..* (or 1..k, k>1), is it possible to have more than one instance associated to the same instances at the other ends?
  \[ G_0 = \text{Assoc}(\{a1, p_1, ..., p_n\}, t) \land \text{Assoc}(\{a2, p_1, ..., p_n\}, t) \land p_i <> p_i \]
  for n-ary associations, where the oid \( a \) is needed in case \( \text{Assoc} \) is an association class, \( p_1, ..., p_n \) are the participants of the association and \( p_i \) is the participant with cardinality 1..*.
- For the case of binary associations included as a term in the predicate representing the class at the other end (as happens in the example) the general goal is:
  \[ G_0 = \text{Class}(c, \text{prop}_1, ..., \text{prop}_i, ..., \text{prop}_n, t) \land \text{class}(c_2, \text{prop}_1, ..., \text{prop}_n, t) \land c_i = c_2 \]
  where \( \text{Class} \) has cardinality 1..* and \( \text{prop}_i \) is the reference to the class with cardinality 1.
- For each cardinality constraint 0..* (or 0..k, k\geq1), is it possible that an instance at the other end does not participate in the association?
  \[ G_0 = \neg \text{Assoc}(\{a, p_1, ..., p_n\}, t) \land \text{class}(p_j, ..., t) \]
  for each \( j <> i \) representing a participant of \( \text{Assoc} \), where \( p_i \) is the participant with cardinality 0..*.
- For binary associations represented as a term:
  \[ G_0 = \text{Class1}(c_1, ..., t) \land \neg \text{Class2}(c_2, ..., c_1, ..., t) \]
  where \( \text{Class1} \) has cardinality 1 and \( \text{Class2} \) has cardinality 0..*.
- For each class hierarchy constrained by \{incomplete\}, is it possible that an instance belongs only to the superclass?
  \[ G_0 = \text{Superclass}(x, ..., t) \land \neg \text{Subclass1}(x, ..., t) \land ... \land \neg \text{SubclassN}(x, ..., t) \]
- For each class hierarchy constrained by \{overlapping\}, is it possible that an instance belongs to more than one subclass simultaneously?
  \[ G_0 = \text{Subclass1}(x, ..., t) \land \text{Subclass2}(x, ..., t) \]
  for each pair of subclasses

In the following cases, the potentially undesirable situation occurs if \( G_0 \) succeeds:

- For each class \( \text{Class} \), does it admit several instances with the same value in all its terms? This may mean that an identifier constraint is missing:
\( G_0 = \text{Class}(c,p_1,\ldots,p_n,t) \wedge \text{Class}(c2,p_1,\ldots,p_n,t) \wedge c<>c2 \)

- For each recursive association \( \text{Assoc} \), can an instance be associated to itself? This may mean that a constraint to guarantee that the association is acyclic or irreflexive, as it is usual in practice, is missing:

\( G_0 = \text{Assoc}(x, x, t) \)

Note that all the proposed questions are situations admitted by the schema. It is not necessary to ask about situations that are prevented by the constraints, such as whether an instance can belong to more than one subclass simultaneously in the presence of a disjointness constraint. This is taken into account in the translation process and we can be sure that this situation will never occur.

Finally, to complete the validation, the designer may ask whichever questions of interest to him or her. By doing this, he or she will be able to determine whether the schema accepts certain instantiations, that can be completely specified with concrete values or can be sample situations expressed in terms of variables. Also, the question may be how to obtain a certain state, and in this case the answer is a sequence of operation calls that result in an IB satisfying a certain condition.

For instance, a question could be “Can the system store bids of unregistered users?”. To this end, the goal that the designer needs to define is:

\( G_0 = \text{Bid}(b,\text{prod},\text{bidder},\text{amount},t) \wedge \text{Unregistered}(\text{bidder},\text{id},\text{email},\text{reason},t) \)

In this case the CQC Method answers positively and gives 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)\}

Another question could be “How can a bid be obtained?” Now the goal is:

\( G_0 = \text{Bid}(b,\text{prod},\text{bidder},\text{amount},t) \)

and the answer of the CQC Method in this case is very similar to the previous one:

\{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)\}

With the answers to these questions the designer will be able to determine if the schema is correctly defined according to the requirements.

5. Related Work

The problem most commonly addressed in the context of ER schemas has been the satisfiability of cardinality constraints [8, 12, 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 [15], in which the schema is translated into logic in order to check properties such as schema satisfiability, liveliness 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 works that deals with the behavioral schema is in the context of deductive databases [7]. The designer can ask how a given state can be reached, but some structural features, such as inheritance, integrity constraints and derivation rules, are not supported by the proposed method.

Still in the same context, [6] 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 methods.

A few proposals address verification and validation in object-oriented conceptual schemas with a behavioral part. In [10] 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 conceptual schemas is HOL-OCL [3]. The method uses a theorem prover to determine some properties on the schema based on equivalence, such as equivalence of two integrity constraints. The theorems to be proved are defined in terms of the meta-model 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 [11], 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.

Summarizing, 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.

6. Conclusions and Further Work

We have proposed a new approach to reason both on the structural and the behavioral parts of conceptual schemas specified in UML with OCL integrity constraints and pre/postconditions.

Our approach allows to automatically determine whether the conceptual schema is correctly defined, through the accomplishment of desirable properties such as strong satisfiability of the schema and applicability and executability of operations; 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.

Our approach consists of two main steps. First, we translate the conceptual schema into a logic representation in such a way that this representation incorporates the effect of operation executions in terms of the instances of classes and associations that are created or deleted. Then, we use an extension of the CQC Method to perform reasoning on the conceptual schema. This extension has been proposed in this paper in order to properly deal with the temporal features of operations.

We have also illustrated the usefulness of our results by applying our approach to a simple conceptual schema and showing the kind of questions we are able to answer.

The main contribution of our work is the ability to perform general reasoning on the behavioral schema, an issue that has not been properly addressed in the past. Moreover, the kind of reasoning we may perform allows checking both that the schema is correct and that it satisfies the requirements.

There are some interesting directions for further work. First, we plan to provide an implementation of the first step of our method, that is, the translation of the conceptual schema into logic. Also, we plan to improve efficiency of the CQC Method when performing this kind of reasoning by taking advantage of the constraints to reduce the search space required to find the solutions.

References

Appendix - Complete Translation of the Example

Integrity constraints:
- Users and Products are identified by their id
  
  context User inv: User.allInstances() -> isUnique(id)
  context Product inv: Product.allInstances() -> isUnique(id)

- The amount of a bid must be greater than the starting price of the product
  
  context Bid inv: self.amount >= self.product.starting-price

- The owner of a product cannot bid for it
  
  context Bid inv: self.bidder->excludes(self.product.owner)

Translation of the Structural Schema

Classes and Associations

Registered(u:id,e-mail,c-card,t) ← AddRegistered(u:id,e-mail,c-card,t2)
  ∧ ¬DeletedRegistered(u,t2,t) ∧ t2 ≤ t

DeletedRegistered(u,t1,t2) ← DelRegistered(u,t) ∧ t0 > t1 ∧ t1 ≤ t2

Unregistered(u:id,e-mail,reason,t) ← AddUnregistered(u:id,e-mail,reason,t2)
  ∧ ¬DeletedUnregistered(u,t2,t) ∧ t2 ≤ t

DeletedUnregistered(u,t1,t2) ← DelUnregistered(u,t) ∧ t0 > t1 ∧ t1 ≤ t2

User(u:id,e-mail,t) ← Registered(u:id,e-mail,c-card,t)
User(u:id,e-mail,t) ← Unregistered(u:id,e-mail,reason,t)

Product(p:id, descr, st-pr, owner, t) ← AddProduct(p:id, descr, st-pr, owner, t2)
  ∧ ¬DeletedProduct(p,t2,t) ∧ t2 ≤ t

DeletedProduct(p,t1,t2) ← DelProduct(p,t) ∧ t0 > t1 ∧ t1 ≤ t2

Bid(b:bidd, prod, amt, t) ← AddBid(b:bidd, prod, amt, t2) ∧ ¬DeletedBid(b,t2,t) ∧ t2 ≤ t

DeletedBid(b,t1,t2) ← DelBid(b,t) ∧ t0 > t1 ∧ t1 ≤ t2
Graphical Constraints (Implicit and Explicit)

OIDs
←Product(p,i1,d1,sp1,o1,t) ∧ Product(p,i2,d2,sp2,o2,t) ∧ i1<>i2
←Product(p,i1,d1,sp1,o1,t) ∧ Product(p,i2,d2,sp2,o2,t) ∧ d1<>d2
←Product(p,i1,d1,sp1,o1,t) ∧ Product(p,i2,d2,sp2,o2,t) ∧ sp1<>sp2
←Product(p,i1,d1,sp1,o1,t) ∧ Product(p,i2,d2,sp2,o2,t) ∧ o1<>o2
←Product(x,i,d,sp,o,t) ∧ User(x,ui,em,t)
←User(u,i1,e1,t) ∧ User(u,i2,e2,t) ∧ i1<>i2
←User(u,i1,e1,t) ∧ User(u,i2,e2,t) ∧ e1<>e2
←User(x,i,e,t) ∧ Bid(x,p,u,a,t)
←Bid(b,u1,p1,a1,t) ∧ Bid(b,u2,p2,a2,t) ∧ u1<>u2
←Bid(b,u1,p1,a1,t) ∧ Bid(b,u2,p2,a2,t) ∧ p1<>p2
←Bid(b,u1,p1,a1,t) ∧ Bid(b,u2,p2,a2,t) ∧ a1<>a2

Hierarchies
←Registered(u,i,e,cc,t) ∧ Unregistered(u,i2,e2,r,t)
←User(u,i,e,t) ∧ ¬IsKindOfUser(u,t)
IsKindOfUser(u,t) ← Registered(u,i,e,c,t)
IsKindOfUser(u,t) ← Unregistered(u,i,e,r,t)

Referential constraints
← Product(p,i,d,sp,u,t) ∧ ¬IsUser(u,t)
← Bid(b,u,p,a,t) ∧ ¬IsProduct(p,t)
IsUser(u,t) ← User(x,i,e,t)
IsProduct(p,t) ← Product(p,i,d,sp,u,t)

Uniqueness of association classes
← Bid(b1,u,p,a,t) ∧ Bid(b2,u,p,a2,t) ∧ b1<>b2

Cardinality constraints
← User(x,i,e,t) ∧ ¬OneProduct(u,t)
OneProduct(u,t) ← Product(x,i,d,sp,u,t)

OCL constraints
← User(u1,i1,e1,t) ∧ User(u2,i2,e2,t) ∧ u1<>u2 ∧ i1=i2
← Product(p1,i1,d1,sp1,u1,t) ∧ Product(p2,i2,d2,sp2,u2,t) ∧ p1<>p2 ∧ i1=i2
← Bid(b1,u1,p1,a1,t) ∧ Product(p2,i2,d2,sp2,u2,t) ∧ a1<sp2
← Bid(b1,u1,p1,a1,t) ∧ Product(p2,i2,d2,sp2,u2,t) ∧ u1=u2

Translation of the Behavioral Schema

Base predicates
RegisterUser(id, email, cc, pid, descr, st, price, t)
UnregisterUser(user, reason, t)
PlaceBid(user, prod, amt, t)
RemoveUser(user, t)
Events are not simultaneous
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{RegisterUser}(i_2, e_2, c_2, p_2, d_2, sp_2, t) \land i_1 \neq i_2\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{RegisterUser}(i_2, e_2, c_2, p_2, d_2, sp_2, t) \land e_1 \neq e_2\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{RegisterUser}(i_2, e_2, c_2, p_2, d_2, sp_2, t) \land c_1 \neq c_2\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{RegisterUser}(i_2, e_2, c_2, p_2, d_2, sp_2, t) \land p_1 \neq p_2\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{RegisterUser}(i_2, e_2, c_2, p_2, d_2, sp_2, t) \land d_1 \neq d_2\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{RegisterUser}(i_2, e_2, c_2, p_2, d_2, sp_2, t) \land sp_1 \neq sp_2\]
\[\leftarrow \text{UnregisterUser}(u_1, r_1, t) \land \text{UnregisterUser}(u_2, r_2, t) \land u_1 \neq u_2\]
\[\leftarrow \text{UnregisterUser}(u_1, r_1, t) \land \text{UnregisterUser}(u_2, r_2, t) \land r_1 \neq r_2\]
\[\leftarrow \text{PlaceBid}(u_1, p_1, a_1, t) \land \text{PlaceBid}(u_2, p_2, a_2, t) \land u_1 \neq u_2\]
\[\leftarrow \text{PlaceBid}(u_1, p_1, a_1, t) \land \text{PlaceBid}(u_2, p_2, a_2, t) \land p_1 \neq p_2\]
\[\leftarrow \text{RemoveUser}(u_1, t) \land \text{RemoveUser}(u_2, t) \land u_1 \neq u_2\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{UnregisterUser}(u_2, r_2, t)\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{PlaceBid}(u_2, p_2, a_2, t)\]
\[\leftarrow \text{RegisterUser}(i_1, e_1, c_1, p_1, d_1, sp_1, t) \land \text{RemoveUser}(u_2, t)\]
\[\leftarrow \text{UnregisterUser}(u_1, r_1, t) \land \text{PlaceBid}(u_2, p_2, a_2, t)\]
\[\leftarrow \text{UnregisterUser}(u_1, r_1, t) \land \text{RemoveUser}(u_2, t)\]
\[\leftarrow \text{PlaceBid}(u_1, p_1, a_1, t) \land \text{RemoveUser}(u_2, t)\]

Creation of instances
\[\text{AddRegistered}(u, i, e, c, t) \leftarrow \text{RegisterUser}(i, e, c, p, d, sp, t) \land \text{oid}(u)\]
\[\text{AddUnregistered}(u, i, e, c, t) \leftarrow \text{UnregisterUser}(u, r, t) \land \text{Registered}(u, i, e, cc, t_2) \land t_2=t-1\]
\[\text{AddProduct}(p, i, d, sp, u, t) \leftarrow \text{RegisterUser}(u, i, e, c, p, d, sp, t) \land \text{User}(u, i, e, c, t) \land \text{oid}(p)\]
\[\text{AddBid}(b, u, p, a, t) \leftarrow \text{PlaceBid}(u, p, a, t) \land \text{Registered}(u, i, e, cc, t_2)\]
\[\land \text{Product}(p, i, d, sp, u, t_2) \land t_2=t-1\]

Deletion of instances
\[\text{DelRegistered}(u, t) \leftarrow \text{UnregisterUser}(u, r, t) \land \text{Registered}(u, i, e, cc, t_2) \land t_2=t-1\]
\[\text{DelRegistered}(u, t) \leftarrow \text{RemoveUser}(u, t) \land \lnot \text{HasProd}(u, t_2) \land \lnot \text{HasBid}(u, t_2) \land t_2=t-1\]
\[\text{DelUnregistered}(u, t) \leftarrow \text{RemoveUser}(u, t) \land \lnot \text{HasProd}(u, t_2) \land \lnot \text{HasBid}(u, t_2) \land t_2=t-1\]
\[\text{HasProd}(u, t) \leftarrow \text{Product}(p, i, d, sp, u, t)\]
\[\text{HasBid}(u, t) \leftarrow \text{Bid}(b, u, p, a, t)\]