Correct Realizations of Interface Constraints with OCL *

Michel Bidoit [1] Rolf Hennicker [2] Françoise Tort [1] Martin Wirsing [2]

[1] Laboratoire Spécification et Vérification, CNRS & ENS de Cachan, France
[2] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany

Abstract. We present an OCL-like formal notation for interface constraints, called ICL, suited to describe the required observable behavior of any correct interface implementation (provided by some class). The semantics of the ICL notation is defined by a translation to the observational logic institution. For specifying constraints on classes we use a subset of OCL to express invariants and pre- and post-conditions on operations. The semantics of the OCL expressions is defined by a translation into an algebraic specification. Using these semantic foundations we introduce a formal correctness notion for implementation relations between interfaces and classes and we show how to prove implementation correctness by using observational proof techniques.

1 Introduction

1.1 Motivations

Pragmatic software engineering methods and graphical notations as provided by UML are an important means to enhance software quality. The correctness of a program w.r.t. a given specification, however, can only be ensured with the help of formal methods. Therefore it is an important aim to investigate possibilities for combining pragmatic and formal approaches to software development.

A seminal step in this direction was provided by the object-oriented language Eiffel where assertions are an integral (and executable) part of class declarations [M88]. Recently, the Object Constraint Language OCL (see [Rat99], [WK99], [KWC99]) offers a formal notation to constrain the interpretation of modeling elements occurring in UML diagrams. OCL is systematically used for rigorous software development in the Catalysis Approach [SW98]. The OCL notation is particularly suited to constrain classes and associations since OCL expressions allow one to navigate along associations and to describe conditions for object attributes in invariants and pre- and post-conditions of the operations. However, OCL can hardly be used to constrain interfaces since at the interface level no attribute and no association either is available. Moreover, OCL is not designed to express implementation relations and, to our knowledge, there is no

* This work is partially supported by the ESPRIT Working Group 29432 CoFI [CoFI] and by the Bayer. Forschungsstiftung.

proof calculi for verifying semantical consequences of constraints and for proving implementations. The aim of this paper is therefore to contribute to a solution to these issues by providing a general method, formal notations and proof techniques which are compatible with OCL and support correct realizations of interfaces with constraints.

1.2 The General Method: an Overview

We focus on UML class diagrams which contain interfaces and corresponding implementations provided by some classes. In order to ensure the correctness of an implementation (also called "realization"), we propose three major steps briefly summarized in the following.

The Interface Level (cf. Section 3). First, the declaration of the operations provided by an interface must be supplemented by a formal description of their semantic properties. For this purpose we adopt a behavioral approach which allows us to specify the observable properties of the operations in an abstract and intuitive way. Technically this is achieved by splitting the operations of an interface into so-called "observer operations" (to be used for observing abstract, non-visible states) and the "other" operations whose behavior can then be specified by describing their effects w.r.t. the given observers. Thus attributes and associations (which are already implementation oriented) are indeed not necessary for constraining the operations. Therefore we propose an interface constraint language, called ICL, with the same syntax as OCL except that the dot-notation of OCL for attributes (and role of association ends) is replaced by a dot-notation for observers. The other changes are that the @pre-notation for attributes is replaced by an @pre-notation for self and that a symbol for expressing "observational equality" of the encapsulated states is added. Semantically, these ideas are realized by translating the invariants and pre- and post-conditions of interface constraints to the observational logic institution [HB99]. The resulting specification is presented in an extension of the algebraic specification language CASL [CoFI98].

The Class Level (cf. Section 4). At the class level we can use attributes and navigations on associations to describe invariants and pre- and post-conditions on the operations. For this purpose we use a subset of OCL which will be supplied with an algebraic semantics by translating OCL class constraints into a corresponding algebraic specification. In particular, this specification provides a precise description of the notion of an object by relating object identifiers and environments. It will also be presented using CASL.

The Implementation Relation (cf. Section 5). Since interface and class constraints are equipped with a well-defined semantics which in each case is given by a class of algebras, it is now straightforward to define formally the correctness of an implementation (realization): we say that a constrained interface I is correctly implemented by a set of classes SC if all algebras belonging to the semantics of (the translation of) SC belong also to the semantics of I. According to the observational semantics of interfaces, this expresses that the implementation has the required observable behavior. For proving the correctness of an implementation we can apply observational proof techniques on the basis of [BH98].[1]

Throughout the presentation we will illustrate our approach with the help of a simple but representative example described in the next section.

2 The Account Example

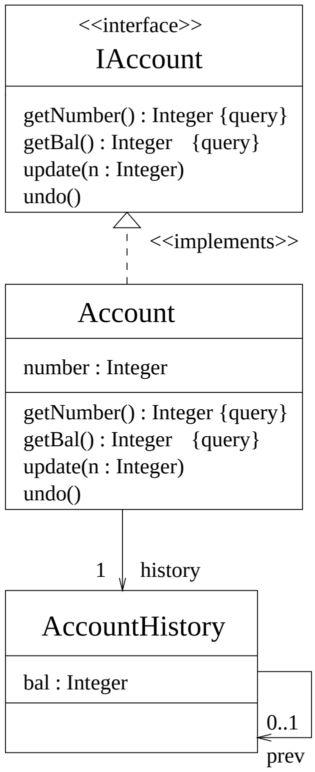

Fig. 1. The UML class diagram for accounts

Figure 1 presents an UML diagram for modeling bank accounts. The IAccount interface shows services that a (simplified) bank account may offer: get the number of the account, get its balance, update the account (with a new

[1] These proof techniques are effective in the sense that they lead to proof obligations that can be discharged using existing first-order theorem provers such as, e.g., the Larch Prover.

positive or negative amount, hence update subsumes the usual credit and debit operations), and undo an update.

A class named Account implements the IAccount interface. Its objects are characterized by a number and a history. The history of an account provides its current balance (the attribute bal of the AccountHistory class) and points to its previous balance (via the association prev).

The expected implementation of the update and undo operations is depicted in Fig. 2, where crosses over links mean that these links are deleted after execution of the operations.

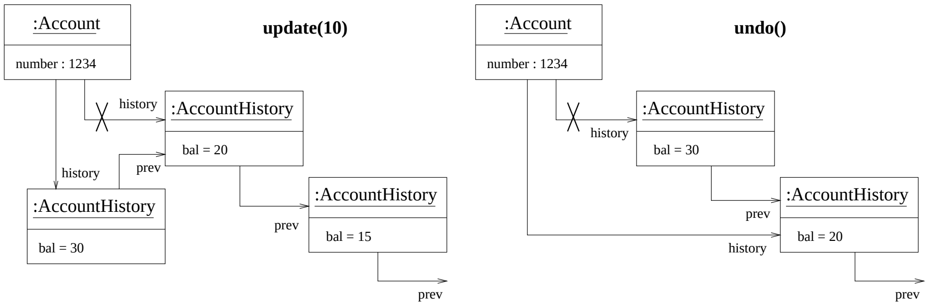

Fig. 2. Impact of update(10) and undo() on links between objects

3 Specification of Interface Constraints

An interface shows the (externally visible) operations of a class. It has no attribute and no association. Three kinds of operations may occur in an interface:

- pure procedures op(x1 : T1, ..., xn : Tn),
- queries op(x1 : T1, ..., xn : Tn) : T, which yield a result and have no side effect (in this case we require the UML property "{query}" to be attached to the operation),
- procedures op(x1 : T1, ..., xn : Tn) : T, which yield a result and may have side effects.

3.1 Interface constraints

In this section, we provide a specification method and a formal notation for interface constraints which allow us to restrict the admissible implementations of an interface to those classes which respect the given interface constraints. Since interfaces do not contain any implementation details, the interface constraints will focus on the "observable behavior" of the operations, i.e., on those properties which are visible "from the outside".

Observer operations. To formally describe observable properties we fix a subset of the interface operations as so-called "observer operations" (or "observers" for short). The behavior of the other operations can then be constrained by defining their effects w.r.t. the observable experiments determined by the given observers. The observers have to be chosen by the programmer in such a way that they adequately express an indistinguishability relation for objects. (This is quite analogous to the functional programming paradigm where the programmer has to choose the adequate data type constructors.) For instance, for the IAccount interface we choose getNumber(), getBal() and undo() as observers.[2] We have chosen these observers because from the user's point of view two accounts a and b are "observationally equal" if they have the same number and the same balance and if this also holds after performing the same number of undos both for a and b (i.e., if a and b have the same history).[3]

Observational specification method. The declaration of a distinguished set of observers leads to a specification method which shows how to constrain the non-observer operations in a systematic way. Thereby the idea is to specify the observable effects of a non-observer operation by a (complete) case distinction w.r.t. the given observers (see [BH99] for a formal treatment of observer complete definitions which indeed are quite analogous to constructor complete definitions in functional programming). For instance, the observable effect of updating an amount for an account is specified w.r.t. the three observers getNumber(), getBal() and undo() (see the IAccount interface example below). Note that in contrast to OCL, we admit the call of observer operations with side effects such as undo(). This is possible since we translate operations into functions and constraints into declarative axioms, hence the constraints are not executed but interpreted in a declarative logical formalism which provides all possibilities for reasoning about state changes without actually performing them (see below the section on the semantics of interface constraints).

Syntax of interface constraints. To express the interface constraints, we use an OCL-like syntax, called ICL, where attributes and roles are replaced by the following constructs:

- Calls of the form o.op(p1,...,pn), where o is an object, op is an observer, possibly with side effects, and p1,...,pn the actual parameters, are allowed. We deviate on purpose in this point from the recommendations in [WK99] for OCL extensions in the sense that observers can change the state of objects. We assume that the interface can address only objects in the arity of the operation op, i.e., either the object o or one of the parameters xi.
- o@pre where o stands for an object. o@pre can be used in a post-condition of an operation to denote the state of o before the execution of the operation. For example, self@pre denotes the (state of the) object self before the call of the

[2] As a graphical notation, the "{observer}" property can be attached to each of these operations in Fig. 1.

[3] undo() is also called an "indirect observer" because it does not directly produce a visible output but nevertheless contributes to distinguish accounts (since after applying an undo(), a "direct observer" getNumber() or getBal() can always be applied).

operation and similarly xi@pre denotes (the state of) an in-out parameter before the call of the operation.
• o ∼ o1 denotes the "observational equality" of the objects o and o1, i.e., the indistinguishability of their respective states.

The IAccount interface is restricted by the following ICL constraints.

```
context IAccount
  inv: self.getBal() >= 0

context IAccount::update(n : Integer)
  pre: self.getBal()+n >= 0
  post: self.getNumber() = self@pre.getNumber() and
        self.getBal() = self@pre.getBal()+n and
        self.undo() ~ self@pre
```

As explained above, in the constraint for update(n), self denotes the current (state of the) account object after applying the update(n) operation and self@pre denotes (the state of the) self before the call of update(n). Moreover, if an update(n) operation followed by an undo() is performed for an account a then the result is not a visible value but rather a new state of the object. Hence the constraint self.undo() ∼ self@pre requires that this state is observationally equal to the original state.

3.2 Semantics of Interface Constraints

In the previous section we have intuitively introduced the syntax of interface constraints and their intended meaning. Formally, the semantics of our interface constraint language ICL is defined by associating an observational specification to a given interface. We do this in two steps: first, we construct an algebraic signature (see [W90]) of a given interface. Then, we translate the invariants and the pre- and post-conditions of the operations to axioms written in many-sorted first-order logic over this signature. For defining the signature of an interface I:
• any pure procedure op(x1 : T1, ..., xn : Tn) is translated into an operation symbol $\_\_.op(\_\_, \ldots, \_\_) : I \times T1 \times \ldots \times Tn \to I$,
• any query op(x1 : T1, ..., xn : Tn) : T is translated into an operation symbol $\_\_.op(\_\_, \ldots, \_\_) : I \times T1 \times \ldots \times Tn \to T$ and
• any procedure op(x1 : T1, ..., xn : Tn) : T with result is translated into an operation symbol $\_\_.op(\_\_, \ldots, \_\_) : I \times T1 \times \ldots \times Tn \to Tuple[I, T]$, whereby $Tuple[I, T]$ is a sort which denotes the cartesian product of I and T.[4]
By exception, if the result type of an operation is Boolean, then the operation is translated into a corresponding predicate symbol.

The atomic formulas built over this signature are either predicates (as usual) or, for terms t and r of the same sort s, equations $t = r$ (if s is the name of a

[4] Here we assume that the operations have only input-parameters. In the case of in-out parameters additional result sorts would be needed.

basic data type like Integer) or observational equations $t \sim r$ (if s is the name of an interface).

For defining the semantics of a constrained interface, we add to the signature the translation of the ICL constraints into observational logic axioms. This translation is done according to the following scheme. Let us consider an interface constraint of the form:

context I::op(x1 : T1, ..., xn : Tn)
  pre: ψ
  post: ϕ

where the context indicates that op is an operation (in this case a pure procedure) of an interface I, x1, ..., xn are input parameters, and ψ and ϕ are logical expressions defining the pre- and post-conditions on op. Let $Trans_{pre}$ and $Trans_{post}$ be functions translating logical expressions of ICL into first-order formulas, then the above constraint is translated into $Trans_{pre}(\psi) \Rightarrow Trans_{post}(\phi)$.

An invariant constraint of the form:

context I
  inv: ϕ

is translated into the axiom $Trans_{pre}(\phi)$.

The "$Trans_{...}$"functions are defined as follows. The ICL boolean operations not, and, or, etc., are translated into the corresponding logical connectives $\neg, \wedge, \vee$, etc. The ICL equality test "=" is kept and considered as an equality symbol to form equations between terms, except for boolean terms where it is translated to logical equivalence $\Leftrightarrow$. The ICL observational equality test "$\sim$" is kept. The ICL operators for collection types are also kept and considered as predefined operation or predicate symbols.

The translation of ICL expressions of the form t.op, where t is a complex expression of the form t1.op1, is recursively defined by $Trans_{...}(\mathsf{t.op}) = Trans_{...}(\mathsf{t}).op$.

The recursion terminates if the translation function is finally applied to a variable which remains unchanged, or if it is applied to self or self@pre. In that cases the translation is more subtle:

- self in an invariant or in a pre-condition denotes the state of the object before the call of the procedure op, it is kept by the translation: $Trans_{pre}(\mathsf{self}) = self$,
- self in a post-condition denotes the state of the object after the call of op, it is translated into $self.op(x1, \ldots, xn)$. Whereas self@pre in a post-condition denotes the state of the object before the call of op, it is translated into $self$: $Trans_{post}(\mathsf{self}) = self.op(x1, \ldots, xn)$ and $Trans_{post}(\mathsf{self@pre}) = self$.

We obtain an observational specification in the framework of the observational logic institution [HB99], whereby the interface constraints are now axioms of the specification. The semantics of such a specification is given by the class of all (observational) algebras which observationally satisfy the axioms. Thereby the observational satisfaction relation is defined in the same way as the usual satisfaction relation of the first order predicate calculus but the symbol "$\sim$" is interpreted by a semantically defined observational equality relation between the elements of an algebra (for details see [HB99]). Based on these semantical notions, one can reuse the sound and complete proof system of [HB99] for proving observational theorems of interfaces with constraints.

The observational specification associated to the IAccount interface is shown below. It is written in an extension of the CASL algebraic specification language [CoFI98] to take into account the distinction between observer and non-observer operations.[5] The specification PREDEFINEDOCLTYPES is an algebraic specification of the basic OCL Types, which is not shown here for lack of space.[6]

obs spec IACCOUNT =
  PREDEFINEDOCLTYPES **then**
  sort $IAccount$;
  obs ops
    $\_\_.getNumber() : IAccount \to Integer$;
    $\_\_.getBal() : IAccount \to Integer$;
    $\_\_.undo() : IAccount \to IAccount$;
  op $\_\_.update(\_\_) : IAccount \times Integer \to IAccount$;
  vars $self{:}IAccount$; $n{:}Integer$;
  axioms
    %% Invariant:
    $self.getBal() \geq 0$;
    %% update:
    $self.getBal() + n \geq 0 \Rightarrow$
      $self.update(n).getNumber() = self.getNumber() \wedge$
      $self.update(n).getBal() = self.getBal() + n \wedge$
      $self.update(n).undo() \sim self$;
end

4 Specification of Class Constraints

4.1 Using OCL to specify Class Constraints

As shown in Fig.1, interfaces are implemented by classes which contain attributes and which may be connected via associations. For describing such implementations we use (a subset) of OCL 1.3 [Rat99] to state invariants and pre- and post-conditions on the operations. Thereby, similarly to the observational specification method, operations can now be constrained by describing their effects

[5] We also make an intensive use of the "display annotation" facility of CASL. This facility allows us to use the ICL syntax for operation names, terms and formulas.

[6] All the sorts introduced by PREDEFINEDOCLTYPES are of course observable ones.

w.r.t. the given attributes and associations (in fact, we will use the role names at the appropriate association ends). For instance, the classes Account and AccountHistory are restricted by the following OCL constraints (which, for update and undo, are conform with Fig. 2).

context AccountHistory
  inv: bal >= 0

context Account::getNumber() : Integer
  post: result = number

context Account::getBal() : Integer
  post: result = history.bal

context Account::update(n : Integer)
  pre: history.bal + n >= 0
  post: number = number@pre and
        history.oclIsNew and
        history.bal = history@pre.bal@pre + n and
        history.prev = history@pre

context Account::undo()
  pre: history.prev→notEmpty
  post: number = number@pre and
        history = history@pre.prev@pre

In the above expressions, we have used certain assumptions and predefined constructs of OCL. First, it is implicitly assumed that attribute and navigation expressions like number, history.bal, history.prev and accounts are shorthand notations for self.number, self.history.bal, self.history.prev and self.accounts where self is a special variable for objects whose type can be derived from the context. Moreover, an expression in a pre-condition refers always to a value before execution of the operation while the same expression in a post-condition refers to the value after execution of the operation. However, one can also refer in a post-condition to a previous value by using the symbol "@pre" at the appropriate positions. For instance, history@pre.bal@pre refers to the previous balance of the previous history object of an account. In a post-condition one can also apply the "oclIsNew" construct to an object o to express that after execution of the operation, o is a new object. For instance, after performing update, a new account history object is created.

4.2 Semantics of Class Constraints

The semantics of the class constraints of a class C is defined by a translation into an algebraic specification C_OO written in the algebraic specification language Casl [CoFI98]. For providing the semantics one has to take into account environments, for which we introduce a sort *Env*, and object identifiers, for which we introduce a sort CId. Then any object can be represented by a pair $<ev, id>$ where ev is an environment and id is an object identifier. However, note that conversely not any pair $<ev, id>$ denotes an object. The reason is that we must always be able to create new objects, i.e., to find an identifier which did not represent an object in an "old" environment ev but does represent an object in a "new" environment ev'. Therefore the resulting specification C_OO introduces, besides the sorts Env and CId, an explicit sort C and a sort $Tuple[Env, CId]$ such that $C < Tuple[Env, CId]$ (i.e., C is a subsort of $Tuple[Env, CId]$). The sort $Tuple[Env, CId]$ (and its corresponding pairing function $<\_\_, \_\_>$ and projections $\_\_.env$ and $\_\_.cId$) is defined by means of the "free-type" construct of CASL to denote the cartesian product of Env and CId; the subsort C is used to denote the subset of all pairs $<ev, id>$ which indeed are objects.

In the next step we introduce appropriate operation symbols which still have to be added to obtain the full signature of the specification C_OO. For this purpose we first associate to any attribute a : T of the class C an operation symbol $\_\_.a : C \rightarrow T$ and for any role r at the association end to a class D an operation symbol $\_\_.r : C \rightarrow DId$ (if the role has at most multiplicity 1) and $\_\_.r : C \rightarrow Set[DId]$ (if the multiplicity of r is greater than 1). Note that we have used the sort DId ($Set[DId]$ resp.) as the result sort of $\_\_.r$ because values of a role are indeed object identifiers. Finally, we introduce for each operation of the class a corresponding operation symbol in the same way as for the operations of interfaces.[7] From a technical point of view it is also convenient to introduce for the attributes, roles and operations, the overloaded operation symbols working on tuple sorts.

The most important step of the translation concerns the translation of OCL constraints into corresponding axioms. There exists a general translation algorithm which (for lack of space) cannot be detailed here. Nevertheless, let us sketch the basic ideas of the translation scheme by considering a constraint of the form:

context C::op(x1 : T1, ..., xn : Tn)
  pre: ψ
  post: ϕ

where op is a pure procedure (the other cases are quite similar). Then ψ refers to an environment ev before execution of op and ϕ refers to an environment ev' after execution of op. Since the constraint must be satisfied whenever op is performed for an object $<ev, self>$ of the class C, we translate the constraint into the following axiom:

$$\begin{array}{l} \forall ev, ev' : Env; self : CId; x1 : T1; \ldots; xn : Tn \bullet \\ \quad <ev, self> \in C \wedge ev' = <ev, self>.op(x1, \ldots, xn).env \wedge Trans_{pre}(\psi) \\ \qquad \Rightarrow <ev, self>.op(x1, \ldots, xn).cId = self \wedge Trans_{post}(\phi) \end{array}$$

The "$\in$" symbol is predefined in CASL to test membership in a subsort and $Trans_{pre}$ and $Trans_{post}$ are functions which translate logical expressions of OCL

[7] Apart from the case where an operation has an object valued parameter. Then the translation is slightly different, but is not explained here for lack of space.

into first-order formulas. The translation of logical expressions and OCL operators is as in ICL (except that now a class C is translated to identifiers of type CId).

Note that a special role is played by the operator $\rightarrow$isEmpty (and similarly by its negation $\rightarrow$notEmpty) which on the one hand can be used to test whether a set is empty (in this case it is translated to the predicate symbol $\_\_\rightarrow isEmpty : Set[CId]$) and, on the other hand, can be used to test whether an object identifier is "nil"(for this test we use the overloaded predicate symbol $\_\_\rightarrow isEmpty : CId$; cf. the pre-condition of undo where the negation $\_\_\rightarrow notEmpty : CId$ is used).[8]

The most interesting case is the translation of OCL "properties" whereby we consider expressions of the form t.a where t is a navigation path (in the simplest case just the variable self) and a is an attribute or a role (at an association end). Then $Trans_{pre}$ and $Trans_{post}$ are recursively defined by:

$$Trans_{pre}(\mathsf{t.a}) = <ev, Trans_{pre}(\mathsf{t})>.a\ ,$$
$$Trans_{post}(\mathsf{t.a}) = <ev', Trans_{post}(\mathsf{t})>.a\ .$$

The recursion terminates if the translation function is finally applied to a variable which remains unchanged. As an example, we compute:[9]

$$\begin{aligned}
&Trans_{post}(\mathsf{history.bal}) = \\
&Trans_{post}(\mathsf{self.history.bal}) = \\
&<ev', Trans_{post}(\mathsf{self.history})>.bal = \\
&<ev', <ev', Trans_{post}(\mathsf{self})>.history>.bal = \\
&<ev', <ev', self>.history>.bal
\end{aligned}$$

The above definition of $Trans_{post}$ is not yet complete because in a post-condition one can also refer to previous values by the symbol "@pre" and one can use the "oclIsNew"construct. In these cases, the translation is defined by:

$$\begin{aligned}
&Trans_{post}(\mathsf{t.a@pre}) = <ev, Trans_{post}(\mathsf{t})>.a\ , \\
&Trans_{post}(\mathsf{t.oclIsNew}) = \\
&\qquad\qquad \neg\,(<ev, Trans_{post}(\mathsf{t})> \in C) \\
&\qquad\qquad \wedge <ev', Trans_{post}(\mathsf{t})> \in C
\end{aligned}$$

Finally let us point out that besides the translated class constraints any specification C_OO still contains two additional axioms, one which requires that "nil is not an object"(expressed by $\rightarrow notEmpty$) and one which requires "consistency of environments". Intuitively, this means that in any fixed environment each navigation path that starts from an object (and does not yield "nil") provides an identifier or a set of identifiers which denote indeed objects in this environment. The following specifications show the result of the translation of the class constraints in our example.

spec ACCOUNTHISTORY_OO =
  PREDEFINEDOCLTYPES **then**

[8] Hence, as in OCL, we do not introduce an explicit denotation for "nil".

[9] This computation is used in the translation of the post-condition of the update operation.

```
  sorts Env, AccountHistoryId;
  free type Tuple[Env, AccountHistoryId] ::=
              <__, __>(__.env : Env; __.accountHistoryId : AccountHistoryId);
  %%  This CASL construct abbreviates the declarations of:
  %%  the sort Tuple[Env, AccountHistoryId], a pairing function <__, __>,
  %%  two projections .env and .accountHistoryId,
  %%  and the corresponding axioms.
  sort  AccountHistory < Tuple[Env, AccountHistoryId];
  ops   __.bal : AccountHistory → Integer;
        __.prev : AccountHistory → AccountHistoryId;
        %% We need also the overloaded versions on the tuple supersort:
        __.bal : Tuple[Env, AccountHistoryId] → Integer;
        __.prev : Tuple[Env, AccountHistoryId] → AccountHistoryId;
  vars  ev:Env; self:AccountHistoryId;
  axioms
        %% "nil" is not an object:
        <ev, self> ∈ AccountHistory ⇒ self→notEmpty;
        %% Consistency of environments:
        <ev, self> ∈ AccountHistory ∧ <ev, self>.prev→notEmpty ⇒
          <ev, <ev, self>.prev> ∈ AccountHistory;
        %% Invariant:
        <ev, self> ∈ AccountHistory ⇒ <ev, self>.bal ≥ 0;
end

spec ACCOUNT_OO =
  ACCOUNTHISTORY_OO then
  sort  AccountId;
  free type Tuple[Env, AccountId] ::=
              <__, __>(__.env : Env; __.accountId : AccountId);
  sort  Account < Tuple[Env, AccountId];
  ops   __.number : Account → Integer;
        __.history : Account → AccountHistoryId;
        __.getNumber() : Account → Integer;
        __.getBal() : Account → Integer;
        __.update(__) : Account × Integer → Account;
        __.undo() : Account → Account;
        %% Overloaded versions on the tuple supersort
        %% omitted for lack of space.
  vars  ev, ev′:Env; self:AccountId; n:Integer;
  axioms
        %% "nil" is not an object:
        <ev, self> ∈ Account ⇒ self→notEmpty;
        %% Consistency of environments:
        <ev, self> ∈ Account ⇒
          <ev, <ev, self>.history> ∈ AccountHistory;
        %% history has multiplicity 1, not 0..1:
```

$$
\begin{array}{l}
\quad <ev, \mathit{self}> \in \mathit{Account} \Rightarrow <ev, \mathit{self}>.\mathit{history}{\rightarrow}\mathit{notEmpty};\\
\quad \text{\%\% getNumber}\\
\quad <ev, \mathit{self}> \in \mathit{Account} \Rightarrow\\
\qquad <ev, \mathit{self}>.\mathit{getNumber}() = <ev, \mathit{self}>.\mathit{number};\\
\quad \text{\%\% getBal}\\
\quad <ev, \mathit{self}> \in \mathit{Account} \Rightarrow\\
\qquad <ev, \mathit{self}>.\mathit{getBal}() = <ev, <ev, \mathit{self}>.\mathit{history}>.\mathit{bal};\\
\quad \text{\%\% update}\\
\quad <ev, \mathit{self}> \in \mathit{Account} \wedge ev' = <ev, \mathit{self}>.\mathit{update}(n).\mathit{env}\\
\qquad \wedge <ev, <ev, \mathit{self}>.\mathit{history}>.\mathit{bal} + n \geq \mathit{0} \Rightarrow\\
\qquad\quad <ev, \mathit{self}>.\mathit{update}(n).\mathit{accountId} = \mathit{self}\\
\qquad \wedge <ev', \mathit{self}>.\mathit{number} = <ev, \mathit{self}>.\mathit{number}\\
\qquad \wedge \neg(<ev, <ev', \mathit{self}>.\mathit{history}> \in \mathit{AccountHistory})\\
\qquad \wedge <ev', <ev', \mathit{self}>.\mathit{history}> \in \mathit{AccountHistory}\\
\qquad \wedge <ev', <ev', \mathit{self}>.\mathit{history}>.\mathit{bal} = <ev, <ev, \mathit{self}>.\mathit{history}>.\mathit{bal} + n\\
\qquad \wedge <ev', <ev', \mathit{self}>.\mathit{history}>.\mathit{prev} = <ev, \mathit{self}>.\mathit{history}\ ;\\
\quad \text{\%\% undo}\\
\quad <ev, \mathit{self}> \in \mathit{Account} \wedge ev' = <ev, \mathit{self}>.\mathit{undo}().\mathit{env}\\
\qquad \wedge <ev, <ev, \mathit{self}>.\mathit{history}>.\mathit{prev}{\rightarrow}\mathit{notEmpty} \Rightarrow\\
\qquad\qquad <ev, \mathit{self}>.\mathit{undo}().\mathit{accountId} = \mathit{self}\\
\qquad\quad \wedge <ev', \mathit{self}>.\mathit{number} = <ev, \mathit{self}>.\mathit{number}\\
\qquad\quad \wedge <ev', \mathit{self}>.\mathit{history} = <ev, <ev, \mathit{self}>.\mathit{history}>.\mathit{prev}\ ;
\end{array}
$$

end

In addition to the above algebraic specifications, which result from our systematic translation of OCL expressions, it is necessary to take into account the so-called “Global Frame Assumption (GFA)” [L95]. For each non-query operation, it is always assumed that the state of any object not explicitely mentioned in the post-condition is left unchanged. This leads to a further algebraic specification SYSTEM_OO, which is just an enrichment of ACCOUNT_OO by the axioms expressing the global frame assumption for *update* and *undo*.

spec SYSTEM_OO =
ACCOUNT_OO **then**
axioms

$$
\begin{array}{l}
\quad \text{\%\% GFA for update:}\\
\quad \forall \mathit{self}{:}\mathit{AccountId};\, ev, ev'{:}\mathit{Env} \bullet\\
\qquad <ev, \mathit{self}> \in \mathit{Account} \wedge ev' = <ev, \mathit{self}>.\mathit{update}(n).\mathit{env} \Rightarrow\\
\qquad\qquad \forall id{:}\mathit{AccountId} \bullet \neg(id = \mathit{self}) \wedge <ev, id> \in \mathit{Account} \Rightarrow\\
\qquad\qquad\quad <ev', id>.\mathit{number} = <ev, id>.\mathit{number}\\
\qquad\qquad\quad \wedge <ev', id>.\mathit{history} = <ev, id>.\mathit{history}\\
\qquad\quad \wedge \forall id{:}\mathit{AccountHistoryId} \bullet <ev, id> \in \mathit{AccountHistory} \Rightarrow\\
\qquad\qquad\quad <ev', id>.\mathit{bal} = <ev, id>.\mathit{bal}\\
\qquad\qquad\quad \wedge <ev', id>.\mathit{prev} = <ev, id>.\mathit{prev}\ ;\\
\quad \text{\%\% Similarly for undo} \ldots
\end{array}
$$

end

5 Justification of the Implementation Relation

We now have all the necessary ingredients to make precise what it means for the classes Account and AccountHistory to be a correct realization of the IAccount interface.

We will say that a set of classes SC is a correct implementation (realization) of an interface I if, by definition, the algebraic specification SYSTEM_OO(SC) which expresses the semantics of the set of classes SC (together with their OCL constraints) is a correct implementation (in the algebraic sense, see, e.g., [ST97] and [BH98]) of the observational specification OBS_SP(I) which expresses the semantics of the interface I (together with its ICL behavioral constraints).

In our example, the classes Account and AccountHistory are a correct implementation of the IAccount interface if the specification SYSTEM_OO is a correct implementation of the observational specification IACCOUNT.

Once we have a formal correctness notion for implementation relations between interfaces and classes, the next step is to provide concrete means for proving the correctness of a given implementation. To do this, we can simply reuse the existing results from the literature (see, e.g., [BH98] and [HB99]). For instance, to prove the correctness of the implementation in our example, it is enough to show that the axioms of IACCOUNT (after renaming *IAccount* into *Account*) are logical consequences of the specification SYSTEM_OO, enriched by a suitable axiomatization of $\sim$. Here, this axiomatization of $\sim$ is as follows:[10]

$$\begin{aligned}
&\forall a, b{:}Account \bullet \\
&\qquad a \sim b \Leftrightarrow (\bigwedge_{i \in \mathbb{N}} a.undo^i.getBal() = b.undo^i.getBal() \\
&\qquad\qquad \wedge\ a.getNumber() = b.getNumber()\ )
\end{aligned}$$

The proof of the correctness of the implementation relation can then be performed using standard proof techniques. Thereby it is indeed essential that we have used an observational approach for interface specifications since the implementation satisfies the required observational equation $a.update(n).undo() \sim a$ but does not satisfy this equation if "$\sim$" is replaced by "="(since update creates a new object).

6 Related Work

There is a large amount of work formalizing class diagrams (see, e.g., [SF97], [EFLR99] using Z, [L95] using Z++ and RAL, [BGHRS97] using stream-oriented algebraic specifications, [WK96] using algebraic specification and rewriting logic, [WB98] using algebraic specification and dynamic logic, [GR98] translating TROLL to UML, and [O98] using operational semantics). Most of these approaches propose also formal notions of implementation (see, e.g., [EFLR99],

[10] After some obvious simplifications. Infinitary conjunctions are not directly expressible in CASL, and a further step, not explained here for lack of space (see [HB99]), is used to get rid of these.

[L95], [WK96]). However, none of these papers treats interfaces directly (although interface constraints can be expressed using dynamic logic or predicates on streams) and none of the papers uses OCL. To our knowledge, also the use of observational logic for expressing the behavior of interfaces is new, although similar semantic approaches such as coalgebra (see, e.g., [JR97]) and hidden algebra (see, e.g., [GM97]) have been used for modeling objects.

There are two other formal semantics of OCL: [RG98] gives a set-theoretical model of OCL with focus on associations and the iterate construct; the semantics of [BG98] is an algebraic semantics using Larch. In particular, the latter approach is similar to our semantics (although both semantics have been developed independently); the main difference is that we give an explicit representation of the set of existing instances as a subsort. In this way we can use the well-known semantic notions and proof methods of algebraic specifications for modeling the implementation of interfaces by classes. In contrast, [BG98] model the set of existing instances implicitly by a boolean function but they are not concerned with (such) implementations. Other differences are the choice of the specification language (Larch instead of Casl) and w.r.t. [RG98] the modeling of undefinedness: [RG98] introduces a bottom element; in [BG98] and also in our approach undefined elements are "unspecified".

7 Concluding Remarks

In this paper we have presented an approach for specifying UML interface constraints and for proving the correctness of implementation relations between interfaces and classes. Our approach includes a proposal for an interface constraint language OCL-like, a method for developing interface constraints and modular proof methods directly derived from the theory of behavioral algebraic specifications. Inheritance has not been considered due to space limitations. It can easily be included e.g. by using the subsorting approach described in [ACZ99]. Moreover, dynamic behavior described by interaction and state diagrams in UML has not been considered in this paper. More generally, an interface is realized by a subsystem consisting of one or several classes whose dynamic behavior is described by sequence, collaboration and state diagrams [JBR99]. It should be straightforward to integrate our class specifications and the OCL semantics into the stream-based system model of [KRB96]. Extending our behavioral interface specifications by sequence diagrams seems to be more difficult since it requires integrating the observational logic of [BH99] with nondeterministic rewriting (for a possible approach see [D96]) or once more with the system model of [KRB96].

Acknowledgment: Thanks go to the anonymous referees for many constructive remarks which helped us to improve the paper.

References

[ACZ99] D. Ancona, M. Cerioli and E. Zucca. *A formal framework with late binding.* Proc. ETAPS/FASE'99, Springer LNCS 1577, pp. 30-44, 1999.

[BG98] M. Brickford and D. Guaspari. *Lightweight analysis of UML*. Draft Technical Report, Odyssey Research Associates, July 1998.

[BH98] M. Bidoit and R. Hennicker. *Modular correctness proofs of behavioural implementations.* Acta Informatica 35:951–1005, 1998.

[BH99] M. Bidoit and R. Hennicker. *Observer complete definitions are behaviorally coherent.* Proc. FM'99 (UGM OBJ/CafeOBJ/MAUDE), to appear, 1999.

[BGHRS97] R. Breu, R. Grosu, F. Huber, B. Rumpe and W. Schwerin. *Towards a precise semantics for object-oriented modeling techniques.* Proc. ECOOP'97 Workshop Reader, Springer LNCS 1357, 1997.

[CoFI] CoFI*: The Common Framework Initiative for algebraic specification and development (WWW pages).* `http://www.brics.dk/Projects/CoFI/`.

[CoFI98] CoFI Task Group on Language Design. Casl – *The* CoFI *algebraic specification language – Summary (version 1.0).* `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`, 1998.

[D96] R. Diaconescu. *Foundations of behavioural specifications in rewriting logic.* Proc. RWLW96, Electronic Notes in Theoretical Computer Science, Vol. 4, 1996.

[EFLR99] A. Evans, R. France, K. Lano and B. Rumpe. *Developing UML as a formal modeling notation.* Proc. The Unified Modeling Language. <<UML>>'98: Beyond the Notation, Springer LNCS 1618, 1999.

[GR98] M. Gogolla and M. Richters. *On combining semi-formal and formal object specification techniques.* Proc. WADT'97, Springer LNCS 1376, pp. 238–252, 1998.

[GM97] J.A. Goguen and G. Malcolm. *A hidden agenda.* Report CS97-538, Univ. of Calif. at San Diego, 1997.

[HB99] R. Hennicker and M. Bidoit. *Observational logic.* Proc. AMAST'98, Springer LNCS 1548, pp. 263–277, 1999.

[JBR99] I. Jacobson, G. Booch and J. Rumbaugh. *The unified software development process.* Reading, Mass.: Addison-Wesley Longman, 1999.

[JR97] B. Jacobs and J. Rutten. *A tutorial on (co)algebras and (co)induction.* EATCS Bulletin 62, pp. 222-259, 1997.

[KWC99] A. Kleppe, J. Warmer and S. Cook. *Informal informality? The Object Constraint Language and its application in the UML metamodel.* Proc. The Unified Modeling Language. <<UML>>'98: Beyond the Notation, Springer LNCS 1618, 1999.

[KRB96] C. Klein, B. Rumpe and M. Broy. *A stream-based mathematical model for distributed information processing systems - Syslab system model.* Proc. FMOODS '96, Chapmann & Hall, 1996.

[L95] K. Lano. *Formal object-oriented development.* Springer, 1995.

[M88] B. Meyer. *Object-oriented software construction.* Prentice Hall International, 1988.

[O98] G. Overgaard. *A formal approach to relationships in the Unified Modeling Language.* Proc. of ICSE'98 (Workshop on Precise Semantics for Software Modeling Techniques). IEEE Computer Society, 1998.

[Rat97] Rational. *Unified Modeling Language: Semantics, Version 1.1.* Rational Software Corporation. `http://www.rational.com/uml/`, 1997.

[Rat99] Rational. *Object Constraint Language, Version 1.3.* Rational Software Corporation. `http://www.rational.com/`, 1999.

[RG98] M. Richters and M. Gogolla. *On formalizing the UML Object Constraint Language OCL.* Proc. 17th Conceptual Modeling – ER'98, Springer LNCS 1507, 1998.

[ST97] D. Sannella and A. Tarlecki. *Essential concepts of algebraic specification and program development.* Formal Aspects of Computing 9:229–269, 1997.

[SF97] M. Shroff and R.B. France. *Towards a formalization of UML class structures in Z.* Proc. COMPSAC'97, IEEE, pp. 646-651, 1997.

[SW98] D. D'Souza and A.C. Wills. *Objects, components and frameworks with UML: the Catalysis approach.* Addison-Wesley, 1998.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language: precise modeling with UML.* Reading, Mass.: Addison-Wesley Longman, 1999.

[WB98] R. Wieringa and J. Broersen. *A minimal transition system semantics for lightweight class- and behavior diagrams.* Proc. of ICSE'98 (Workshop on Precise Semantics for Software Modeling Techniques). IEEE Computer Society, 1998.

[W90] M. Wirsing. *Algebraic specification.* Handbook of Theoretical Computer Science, North-Holland, pp. 675-788, 1990.

[WK96] M. Wirsing and A. Knapp. *A formal approach to object-oriented software engineering.* Proc. RWLW96, Electronic Notes in Theoretical Computer Science, Vol. 4, 1996.