On the Precise Meaning of OCL Constraints

Rolf Hennicker[1], Heinrich Hussmann[2], and Michel Bidoit[3]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany
hennicke@informatik.uni-muenchen.de
[2] Fakultät Informatik, Technische Universität Dresden, Germany
hussmann@inf.tu-dresden.de
[3] Laboratoire Spécification et Vérification, CNRS & ENS de Cachan, France

Abstract. When OCL is applied in concrete examples, many questions arise about the precise meaning of OCL constraints. The same kind of difficulties appears when automatic support tools for OCL are designed. These questions are due to the lack of a precise semantics of OCL constraints in the context of a UML model. The aim of this paper is to contribute to a clarification of several issues, like interpretation of invariants and pre- and postconditions, treatment of undefined values, inheritance of constraints, transformation rules for OCL constraints and computation of proof obligations. Our study is based on a formal, abstract semantics of OCL.

1 Introduction

The Object Constraint Language (OCL) is a part of the UML standard that can be used for several different purposes with quite different goals. One key application is to provide precise information in the definition of standards, like the UML standard itself. The main focus of this paper is, however, on software development, where OCL can be used for precise specification of constraints on the model level. Applying OCL to software specification has a strong potential to improve software quality and software correctness. The rules expressed in OCL can be used by advanced support tools, for instance in the following ways:

- To check the validity of constraints at system runtime, for instance by generating assertion code from OCL [13] or by using the integrity checking mechanisms of database systems [5, 6].
- To experiment with symbolic representations of possible object configurations, as a means to check consistency and correctness of the rule set [17].
- To statically prove that the code never violates the constraints [11].

This paper is motivated by the experiences the authors made when preparing OCL examples for teaching purposes and when designing support tools for OCL. As soon as the technical details of OCL/UML are considered seriously, a number of semantic questions arise which cannot be answered properly by referring to the existing documentation. There is a clear lack of precise semantics for OCL constraints on a UML model.

However, this paper neither intends to give one more semantics for UML in general, nor for the core expression language of OCL. There is already a quite satisfactory definition for the evaluation of an OCL expression on top of an object configuration (a "snapshot" of the model), see [16]. We are addressing the "grey zone" in between UML and OCL. There is not much material yet of this kind, neither in the literature nor in the standard, which explains the interpretation of a constraint in the context of a UML model.

Another observation motivating this work is that the UML standard (draft version 1.4) [18], as far as it deals with pre- and postconditions, appears to be inconsistent with most of the scientific literature on pre- and postconditions (see Section 3 below). Here is a more comprehensive list of similar issues which need to be clarified:

- At which point in the execution of the program is the validity of an invariant enforced?
- What is the meaning of a precondition attached to an operation (without a postcondition)?
- What happens if the precondition of an operation is violated?
- What is the meaning if several constraints are attached to the same operation?
- In howfar do the constraints of a superclass have an impact on the constraints of its subclasses? How is this related to the Liskov substitution principle?
- Is it possible to specify potentially non-terminating operations with OCL? What is the meaning of a constraint in the non-termination case?

In this paper, we elaborate on each of these questions, discuss the range of possible answers and try to find out a "standard" way to answer (or exclude) the questions in a way which makes the practical construction of support tools feasible. Of course, some of the material in this paper can also be understood as a suggestion for improving the UML specification.

This paper is structured as follows: In Sections 2 through 5, an informal discussion of the semantic issues from above is carried out using a simple example. Section 2 deals with invariants, and Section 3 with pre- and postconditions. Section 3 introduces a number of semantic variations for the precise meaning of pre- and postconditions. In Section 4, we develop a simple theory of splitting and combining OCL constraints, which takes into account these semantic variants. Section 5 addresses the specific questions raised by inheritance. Section 6, finally, provides an abstract semantics for UML/OCL, which serves as a mathematical justification of the definitions and decisions taken further above. With the only exception of Section 6, the use of mathematical notation is mostly avoided in this paper, in order to reach a large audience. We assume, however, some basic familarity of the reader with UML and OCL.

2 Invariants

The most prominent use of OCL constraints is for defining *invariants*. An invariant is a very declarative way to specify precise constraints for a class diagram.

It defines addititional rules which have to be obeyed in any object configuration (object diagram) which is constructed for the actual class diagram.

The following example class diagram for bank accounts is used as a running example throughout the rest of this paper.

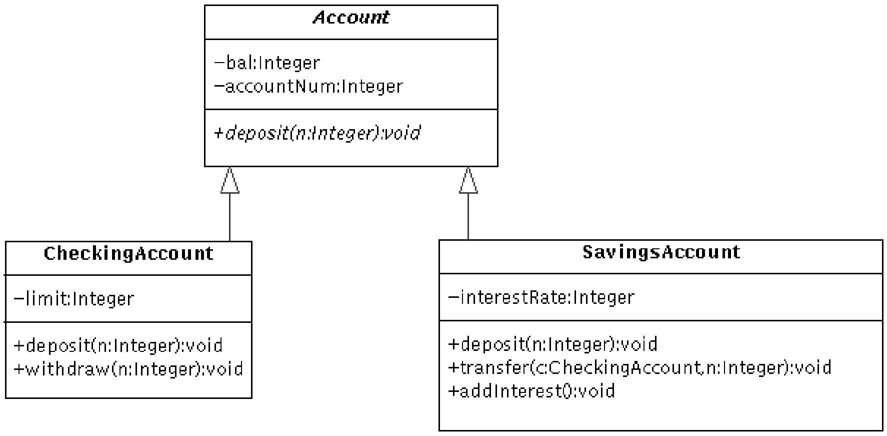

Fig. 1. Example Class Diagram for Accounts

Based on this class diagram, the following example invariant states that for any instance of a checking account, the balance of the account should never go below a given limit.

```
context CheckingAccount
inv: bal >= limit
```

2.1 Checkpoints

In the book on OCL by the main authors of the language (Jos Warmer, Anneke Kleppe), the following simple explanation for the meaning of an invariant is given:

> An *invariant* is a constraint that states a condition that must always be met by all instances of the class, type or interface. [19]

In a more recent errata list for the book, a more moderate text is suggested:

> Invariants must be true upon completion of the constructor and every public method but not necessarily during the execution of methods. [20]

The reason for the changed formulation is obvious: Invariants are often violated during intermediate computation steps. There should exist specific "checkpoints" during the computation at which invariants are enforced. As an example, it may be the case that during the processing of a series of money transfers the balance goes temporarily below the given limit, but comes back above the limit at the end of the overall operation. If the checkpoint is at the end of the whole series of transfers, this should not be treated as a violation of the constraint.

As suggested by Warmer and Kleppe we consider the end of the execution of each *public* operation and of each constructor as such a checkpoint. But what about private and protected operations? In general such operations may be used as auxiliary operations to compute intermediate results. Then they should not be forced to respect an invariant. For this purpose we use in this paper a property named *volatile* (with a Boolean value) such that the tagged value $\{volatile = true\}$ (abbreviated by $\{volatile\}$) can be attached to a non-public operation. This indicates that the operation does *not* require validity of the invariants at the end of its execution.

2.2 Proposed Informal Semantics

Given these clarifications, the intuitive meaning of an invariant can be defined as follows (taking into account additionally the possibility of an undefined result):

> **Informal semantics of invariants:**
> An invariant is a predicate on object configurations (snapshots). Any constructor delivers an object configuration in which the invariant is valid. Moreover, if a public operation or a non-public operation with property $volatile = false$ is applied to an object configuration where the invariant holds, and delivers a defined result, then the invariant also holds for the resulting object configuration.

3 Pre- and Postconditions

As soon as actual transformation of the system state is addressed, *preconditions* and *postconditions* are used. Pre- and postconditions specify operations in a descriptive, non-operational way by comparing the states before and after the execution of the operation. Before discussing the semantics of pre- and postconditions, we have to touch one important general issue.

3.1 Undefinedness

In general, the evaluation of an operation may either yield a defined or an undefined result. The “undefined” result is not a proper value but a pseudo-value indicating the non-existence of a result. In practice, there is an important distinction between two kinds of undefinedness:

– *Exception undefinedness:* In this case, the undefined result can just be considered as a special value different from all defined results. Classical examples are division by zero in integer arithmetic, or an attempt to access an object attribute through a reference which is *null*. It is relatively simple to equip

programs in such a way that they can deal with this kind of non-values, e.g. by exception handling.
- *Non-termination undefinedness:* In this case, the program goes into an infinite loop and refuses to deliver a result. This kind of undefinedness is much more difficult to handle, since it cannot be easily detected, neither statically nor at runtime. Exception mechanisms are useless here.

OCL provides special treatment for undefinedness. The language was explicitly designed in such a way that the evaluation of an OCL query on a snapshot always terminates. This can be seen from the fact that recursion in explicit definition of query operations is required to always terminate (unfortunately a property which is statically undecidable). So in the interpretation of an OCL expression we have to deal only with what was called "exception undefinedness" above.

The treatment of undefinedness in OCL leads into the unpleasant situation that OCL currently uses a three-valued propositional logic for its *Boolean* sort (with values $true$, $false$ and $undefined$). This makes handling of Boolean expressions counter-intuitive. For instance, the well-known law from propositional logic

$$A \Rightarrow B = \neg A \vee B$$

is no longer valid in standard OCL and has to be replaced by the rule

$$A \Rightarrow B = \neg A \vee (A \wedge B)$$

At this point, we do not follow the OCL standard in this paper. In order to enable traditional argumentation on the level of propositional logic, we assume classical two-valued logic. We strongly suggest that the standard is aligned in the same direction, for instance by regarding an undefined value in an evaluation as equivalent to $false$. Using a special logic in OCL will cause major problems in applying existing theorem proving systems, for instance.

Finally, it should be pointed out that the restriction to exception undefinedness in OCL expressions does not at all mean that all specified operations have to terminate. We require here, as OCL does, that all query operations used in OCL expressions are terminating. But other operations transforming the state may terminate or not. (Requiring also those operations to terminate would lead to a quite limited applicability range of the UML/OCL specification language.)

3.2 Semantic Variations

The discussion of undefinedness will turn out as useful for defining the semantics of pre- and postconditions. Let us start with a simple example of a precondition and a postcondition (in the context of the class diagram from above).

```
context Account::deposit(n: Integer)
pre: n >= 0
post: bal = bal@pre + n
```

The central idea of a pre/postcondition pair is a *contract* between the operation and its context. If the operation can rely on the fulfillment of its precondition, then it guarantees its postcondition. This concept is based on a solid theory of program correctness which goes back to the late Sixties ([9], [12]). There are two different classical definitions for the semantics of pre/postconditions which differ in their treatment of undefinedness.

Partial Correctness
An operation $op(x)$ is called *partially correct* with respect to a pair of a precondition PRE and a postcondition $POST$ if the following property holds: If the precondition PRE is valid for a given snapshot σ of the system and if additionally the application of op to σ leads to a defined result, then the postcondition $POST$ is valid for the resulting system state.

Total Correctness
An operation $op(x)$ is called *totally correct* with respect to a pair of a precondition PRE and a postcondition $POST$ if the following property holds: If the precondition PRE is valid for a given snapshot σ of the system, then the application of op to σ leads to a defined result and the postcondition $POST$ is valid for the resulting system state.

The difference between the two definitions is quite subtle. To put it simply, partial correctness does not impose any definedness requirements, whereas total correctness defines a contract which includes definedness of the result.

For an example, let us assume we have an object a of class *Account* with $balance = 0$. Moreover, we apply the operation $deposit(5)$ to a. The difference between the two semantic variations is as follows:

- In partial correctness semantics, it is allowed that $a.deposit(5)$ does not terminate. But when it terminates, the balance after the operation has to have the correct value 5.
- In total correctness semantics, $a.deposit(5)$ has to be defined and the balance after the operation has to have the correct value 5.

Let us now try a more elaborate case. We apply the operation $deposit(-5)$ to a. The average programmer, we are sure, would assume that the precondition $(n \geq 0)$ somehow excludes this application. But the two different definitions do *not* say anything about this application. They only cover the case when the precondition is valid, which is not the case here. So admitted implementations are, among others:

- $a.deposit(-5)$ may raise an exception.
- $a.deposit(-5)$ may go into an infinite loop.
- $a.deposit(-5)$ may just leave the balance untouched.
- $a.deposit(-5)$ may subtract 5 units from the balance.
- $a.deposit(-5)$ may subtract an arbitrary number of units from the balance.

From a practical point of view, it would be very helpful to have some definition which excludes at least the third and fourth implementation!

It is interesting to look at the UML literature for the opinions of the authors regarding the interpretation of pre- and postconditions. The UML standard [18] says that a precondition "must hold for the invocation of an operation" and that a postcondition "must hold after the invocation of the operation". This excludes all the unpleasant effects from above, but does not correspond to the classical definitions on program correctness. Secondary literature on UML is incosistent with the standard. So [3] says (p. 125): "The meaning of the precondition in UML ... is often misinterpreted. [...] The precondition is *not* the operation under which the operation is called. [...] The precondition is the condition under which the operation guarantees that the postcondition will be true." In [7], even two different styles of semantics are distinguished (p. 121), called "$pre \Longrightarrow post$" and "$pre\&post$". The reason for using the first style of semantics is that it is easier to collect independent requirements in this style.

At this point it should be very clear that a formal framework for discussing the variants of pre/postcondition semantics in UML is helpful. The classical definitions from above, however, do not yet cover what the intention of the UML standard seems to be. But there exist other definitions in the literature. In the context of the VDM specification language [8], the support tools offer possibilities to additionally check at each operation invocation whether the precondition is fulfilled. Similarly, in the Eiffel language [15] the notion of class correctness is based on partial (or total) correctness while assertion checks can be optionally performed at runtime. An explicit statement of a clear semantics of parameter restriction is found in the results of the CIP project [4]. The idea used there for precondition semantics (called "parameter restrictions") is the following:

> If the precondition of an operation has not held at invocation time, the result of the invocation is undefined. (Or, equivalently: If the result of an operation is defined, the precondition has held at invocation time.)

Combining partial correctness with this idea gives another semantic variation. We call this semantic variation "partial exception correctness", since it somehow enforces an exception to be raised when the precondition is violated.

> **Partial Exception Correctness**
> An operation $op(x)$ is called *partially exception-correct* with respect to a pair of a precondition PRE and a postcondition $POST$ if the following two properties hold:
> – The operation op is partially correct with respect to PRE and $POST$.
> – If the application of op to σ leads to a defined result, then the precondition PRE has held at invocation time.

The application of this definition to the example shows a clear improvement from the practicians' point of view: The result of $a.deposit(-5)$ has to be undefined. But still, partial expection correctness does not exclude implementations where the result of "normal cases" like $a.deposit(5)$ is undefined.

A fourth semantic variation finally tries to combine total correctness with the idea of parameter restriction:

Total Exception Correctness
An operation $op(x)$ is called *totally exception-correct* with respect to a pair of a precondition PRE and a postcondition $POST$ if the following two properties hold:
- The operation op is totally correct with respect to PRE and $POST$.
- The application of op to σ leads a defined result if and only if the precondition PRE has held at invocation time.

An interesting observation is that proper semantical definitions can be given only for pairs of pre- and postconditions. However, a missing pre- or postcondition can always be given the default value *true*. So an interesting issue is how constraints can be in a correct way merged into pairs or splitted into parts.

4 Splitting of Constraints

In [19] and in the UML standard, it is suggested that complex pre- and postconditions should be splitted into smaller ones. It is obvious that splitting (and merging) of constraints is very helpful from a methodological point of view. In this section we discuss the four semantic variants under the aspect whether they are compatible with splitting (and merging) of constraints and we present transformation rules which allow us to combine constraints. The soundness of the transformation rules can be formally proven by using the semantics approach presented in Section 6.

Splitting of constraints means to specify several pre- and postcondition constraints for the *same* operation. For instance, using our example, we could require the following two constraints, in the following called $Cons1$ and $Cons2$, for the operation *withdraw* of $CheckingAccount$. Please note that $Cons1$ is an example of a stand-alone precondition as it is admitted in UML, extended with a trivial postcondition.

```
context CheckingAccount::withdraw(n: Integer)      -- Cons1
pre: n >= 0
post: true

context CheckingAccount::withdraw(n: Integer)      -- Cons2
pre: bal - n >= limit
post: bal = bal@pre - n
```

Intuitively, this means that both constraints should be satisfied by any realization of *withdraw*. Then it is an obvious question whether the above two constraints are equivalent to the following combined constraint, called $Cons3$.

```
context CheckingAccount::withdraw(n: Integer)      -- Cons3
pre: (n >= 0) and (bal - n >= limit)
post: bal = bal@pre - n
```

In the following, we distinguish the four different semantic variations studied in the previous section.

4.1 Splitting and Partial Correctness

If we consider partial correctness, in fact, $Cons1$ is an empty requirement. Then the question is whether $Cons2$ and $Cons3$ are equivalent. But it is obvious that $Cons2$ is indeed stronger than $Cons3$. Consider, for instance, an object ca of $CheckingAccount$ in the state $bal = 0, limit = 0$ and perform $ca.withdraw(-2)$. Assume that $ca.withdraw(-2)$ is defined. The precondition of $Cons2$ is satisfied and hence $Cons2$ requires that in the resulting state $bal = 2$ holds. Obviously, this is not required by $Cons3$ since the precondition of $Cons3$ is violated.

However, nevertheless there exists a general transformation rule which allows us to combine OCL constraints in the following way:

Transformation Rule for Partial Correctness

```
context C::op(x: T)                  context C::op(x: T)
pre: PRE1 post: POST1                pre: PRE2 post: POST2
```

The two above constraints are equivalent to

```
context C::op(x: T)
pre: true
post: (PRE1@pre implies POST1) and (PRE2@pre implies POST2)
```

Here and in the following, for an OCL expression E which does not contain $@pre$, $E@pre$ denotes the OCL expression obtained from E by adding $@pre$ to all attributes and queries occurring in E.

4.2 Splitting and Total Correctness

The semantics of the first OCL constraint $Cons1$ already entails that the applications $ca.withdraw(n)$ are defined for any $CheckingAccount$ ca, as soon as n is not negative. There is no freedom left for the implementor to leave any special case undefined which has a non-negative value of n. This, however, is allowed in the combined version $Cons3$ of the two OCL constraints. So, again the equivalence of the considered constraints fails.

However, also in the case of total correctness, there exists a general transformation rule which allows us to combine OCL constraints. The idea is very simple: Instead of using an *and* the two preconditions must be combined with an *or* which leads to the following rule.

Transformation Rule for Total Correctness

```
context C::op(x: T)                  context C::op(x: T)
pre: PRE1 post: POST1                pre: PRE2 post: POST2
```

The two above constraints are equivalent to

```
context C::op(x: T)
pre: PRE1 or PRE2
post: (PRE1@pre implies POST1) and (PRE2@pre implies POST2)
```

4.3 Splitting and Partial Exception Correctness

In the case of partial exception correctness the good news is that indeed the transformation works as expected. Here we obtain the following general transformation rule.

Transformation Rule for Partial Exception Correctness

```
context C::op(x: T)                    context C::op(x: T)
pre: PRE1 post: POST1                  pre: PRE2 post: POST2
```

The two above constraints are equivalent to

```
context C::op(x: T)
pre: PRE1 and PRE2
post: POST1 and POST2
```

4.4 Splitting and Total Exception Correctness

Finally, let us consider total exception correctness. Here the bad news is that it doesn't work properly with respect to splitting of constraints. For a counterexample, assume that we want to withdraw 2 units, i.e. we are considering the application $ca.withdraw(2)$. The definition says now together with the first OCL constraint $Cons1$ (precondition $n \geq 0$) that the result of $ca.withdraw(2)$ has to be defined (since $2 \geq 0$). The second OCL constraint $Cons2$ now implies that $bal - 2 \geq limit$ (since the application $ca.withdraw(2)$ is defined). Since, this is a completely independent constraint, this holds for arbitrary object configurations, including those where e.g. $bal = 1$ and $limit = 0$. This is a contradiction, since $1 - 2 >= 0$ does not hold.

In fact, in this case, the combination of two OCL-constraints only works if the two preconditions are equivalent. In all other cases one will end up with an inconsistent specification.

5 Inheritance of Constraints

When using constraints in the presence of inheritance relations a principle goal is to respect Liskov's Substitution Principle which says:

> Wherever an instance of a class is expected, one can always substitute an instance of any of its subclasses [14].

There are two possible approaches how to deal with this requirement. One possibility is to put the responsibility on the system engineer who has to guarantee that the constraints used in his/her model satisfy the substitution principle. Several rules which achieve this goal, like strengthening of invariants in subclasses, are defined in [19]. The disadvantage of this approach is not only the risk that a system engineer may have overlooked some critical cases but also that any constraints imposed on a superclass, must either be redefined or simply be repeated for any subclass. This treatment of constraints seems also to be not consistent with the object-oriented paradigm (where attributes and operations of a superclass belong automatically to all its subclasses).

The second approach, which we find much more convenient, is to consider all constraints imposed on superclasses as implicit constraints for all of its subclasses. To illustrate this simple idea, let A be a superclass of a class B and assume that the following constraints are given:

```
context A inv: INV1
context A::op(x: T)
pre: PRE1
post: POST1
```

```
context B inv: INV2
context B::op(x: T)
pre: PRE2
post: POST2
```

Then the complete set of (explicit) constraints for B is computed as follows:

```
context B inv: INV1
context B::op(x: T)
pre: PRE1
post: POST1
```

```
context B inv: INV2
context B::op(x: T)
pre: PRE2
post: POST2
```

According to the transformation rules of the last section one may still simplify these set of constraints. For instance, if we consider partial exception correctness the completion of the constraints for B could be represented by

```
context B inv: INV1 and INV2
context B::op(x: T)
pre: PRE1 and PRE2  post: POST1 and POST2
```

In particular, when dealing with refinement relations and correctness notions for implementations it is important to know which are the complete requirements (or proof obligations) of a given class or class diagram (with constraints). For this the completion mechanism from above is appropriate which computes stepwise the complete set of requirements for each class of a given class hierarchy.

6 Abstract Semantics

The aim of this section is to provide a semantic foundation for the concepts considered in the previous sections. We do not intend to present a detailed technical approach but we rather want to provide an abstract semantics, based on set-theory, which is as simple as possible to capture our intuition about the precise meaning of OCL constraints. In contrast to other semantic approaches (cf. e.g. [2, 16, 1]) we do not propose a fixed interpretation but we formalize the different alternatives for the interpretation of constraints according to the various correctness notions for pre- and postconditions discussed in Section 3.

In the following we will always assume that $\mathcal{C}$ denotes a UML class diagram and if C is a class occuring in $\mathcal{C}$ we write $C \in \mathcal{C}$.

6.1 Object Configurations

For any class diagram $\mathcal{C}$ there exists a set of object configurations which represent the possible states of the system. These states are also called "snapshots". They can be graphically depicted by UML object diagrams. A detailed semantics of object configurations is given in [16]. We will use the following notations:

- $State(\mathcal{C})$ = set of possible object configurations (snapshots) of $\mathcal{C}$.
- For each $C \in \mathcal{C}$, $Id(C)$ = countably infinite set of potential object identifiers of type C.
- For each $C \in \mathcal{C}$ and $\sigma \in State(\mathcal{C})$, $Instances(C, \sigma)$ = finite subset of $Id(C)$ consisting of the existing objects of type C in the state σ.

6.2 Interpretation of Operations

The set $State(\mathcal{C})$ is solely determined by the associations and attributes occuring in the class diagram $\mathcal{C}$. To give an interpretation for the operations of the classes in $\mathcal{C}$ we use state transformation functions.

For simplicity, we will consider here only unary operations of the form $C :: op(x : T)$ where T is an object type (i.e. a class name) and C is a class with an operation $op(x : T)$ [1]. We also assume that op is not a constructor which would need a special, though not difficult, treatment. Then an interpretation of op is a partial state transformation function

$$I(C :: op(x : T)) : State(\mathcal{C}) \times Id(C) \times Id(T) \to State(\mathcal{C})$$

where $State(\mathcal{C}) \times Id(C) \times Id(T) = \{(\sigma, i, j) \mid \sigma \in State(\mathcal{C}), i \in Id(C), j \in Id(T) \; such \; that \; i \in Instances(C, \sigma) \; and \; j \in Instances(T, \sigma)\}$

This means that $I(C :: op(x : T))$ can be applied to a state σ, to an existing object i of class C (representing an interpretation of $self$) and to an existing object j of class T (representing an interpretation of x). The result $I(C :: op(x : T))(\sigma, i, j)$ represents the new state after execution of op.

6.3 Semantics of Class Diagrams

To provide a semantics of a class diagram $\mathcal{C}$ we need (simultaneous) interpretations for all operations occurring in $\mathcal{C}$. For this purpose let

$$Opns(\mathcal{C}) = \text{set of all operations occuring in classes of } \mathcal{C}$$

Then we consider functions

$$I : Opns(\mathcal{C}) \to StateTransFunct$$

[1] The generalization of our approach to arbitrary many arguments of arbitrary types and to operations with results is straightforward.

where $StateTransFunct$ denotes the set of partial state transformation functions on $State\mathcal{C})$ with appropriate functionality as indicated in Section 6.2 above.

Since, in the moment, we do not consider constraints, arbitrary interpretations of the operations of $\mathcal{C}$ are possible. Hence, the semantics of a class diagram $\mathcal{C}$ is given by the pair

$$Sem(\mathcal{C}) = (State(\mathcal{C}), \{I : Opns(\mathcal{C}) \to StateTransFunct\})$$

In the next step, we will attach OCL constraints to $\mathcal{C}$ which, from the semantical point of view, simply means to restrict (i.e. constrain) the set {I : Opns(C) → StateTransFunct } to those interpretation functions which satisfy the given OCL constraints.

6.4 Satisfaction of OCL constraints

In this subsection we define a satisfaction relation between interpretation functions and OCL constraints. First, let us consider the satisfaction of pre- and postconditions. In fact, corresponding to the four different correctness notions discussed in Section 3, we consider four different kinds of satisfaction relations, denoted by $\models_{pc}$, $\models_{tc}$, $\models_{pec}$ and $\models_{tec}$, which reflect partial correctness, total correctness, partial exception correctness and total exception correctness respectively. As above we will, for simplicity, consider unary operations of the form $C :: op(x : T)$ which are now equipped with OCL constraints

```
context C:: op(x:T)
pre: PRE  post: POST
```

Thereby, PRE and $POST$ are OCL expressions of type $Boolean$ containing (at most) the free variables $self$ and x. We write $PRE_{\sigma,v}$ for the interpretation of PRE in a state $\sigma \in State(\mathcal{C})$ w.r.t. a valuation v such that $v(self) \in Instances(C, \sigma)$ and $v(x) \in Instances(T, \sigma)$. Since POST can contain expressions which refer to an "old" state and, as well, expresions which refer to the "new" state (after execution of op) by using the $@pre$ construct we need two states σ and σ' for its interpretation which is denoted by $POST_{\sigma,\sigma',v}$.

In general PRE and $POST$ may also contain queries. Then, we need additionally an interpretation function I on $Opns(\mathcal{C})$ (cf. above) and we write $PRE_{\sigma,v,I}$ and $POST_{\sigma,\sigma',v,I}$ for the corresponding interpretations. We have now the necessary prerequisites to define the satisfaction relations.

Definition 1. *Let $I : Opns(\mathcal{C}) \to StateTransFunct$ be an interpretation function.*

1. ***Satisfaction w.r.t. partial correctness***
   $I \models_{pc}$ `context C :: op(x : T) pre : PRE post : POST`
   if for all $(\sigma, i, j) \in State(\mathcal{C}) \times Id(C) \times Id(T)$ the following holds:
   if $PRE_{\sigma,v,I} = true$ and $I(op)(\sigma, i, j)$ is defined then $POST_{\sigma,I(op)(\sigma,i,j),v,I} = true$ (where $v(self) = i, v(x) = j$). [2]

[2] $I(op)$ abbreviates $I(C :: op(x : T))$.

2. ***Satisfaction w.r.t. total correctness***
$I \models_{tc}$ `context C :: op(x : T) pre : PRE post : POST`
if for all $(\sigma, i, j) \in State(\mathcal{C}) \times Id(C) \times Id(T)$ *the following holds:*
if $PRE_{\sigma,v,I} = true$ *then* $I(op)(\sigma, i, j)$ *is defined and* $POST_{\sigma,I(op)(\sigma,i,j),v,I} = true$ *(where* v *is as above).*
3. ***Satisfaction w.r.t. partial exception correctness***
$I \models_{pec}$ `context C :: op(x : T) pre : PRE post : POST`
if for all $(\sigma, i, j) \in State(\mathcal{C}) \times Id(C) \times Id(T)$ *the following holds:*
if $I(op)(\sigma, i, j)$ *is defined then* $PRE_{\sigma,v,I} = true$ *and* $POST_{\sigma,I(op)(\sigma,i,j),v,I} = true$ *(where* v *is as above).*
4. ***Satisfaction w.r.t. total exception correctness***
$I \models_{tec}$ `context C :: op(x : T) pre : PRE post : POST`
if for all $(\sigma, i, j) \in State(\mathcal{C}) \times Id(C) \times Id(T)$ *the following holds:*
$I(op)(\sigma, i, j)$ *is defined if and only if* $PRE_{\sigma,v,I} = true$ *and in this case* $POST_{\sigma,I(op)(\sigma,i,j),v,I} = true$ *(where* v *is as above).*

Let us now consider invariants attached to classes $C \in \mathcal{C}$. As discussed in Section 2 an invariant

```
context C inv: INV
```

requires that all public operations and all non-public operations *op* of $\mathcal{C}$ which have the property {volatile=false} respect the invariant INV. More precisely, this means that if INV is satisfied in some state σ and if in this state *op* is defined and yields the new state σ' then INV holds in σ'. Obviously, this expresses just partial correctness w.r.t. INV as pre- and postcondition.

Definition 2. *Let* $I : Opns(\mathcal{C}) \to StateTransFunct$ *be an interpretation function.*

1. ***Satisfaction of invariants***
$I \models$ `context C inv : INV`
if $I \models_{pc}$ `contextC :: op(...) pre : INV post : INV` *for all public operations and all non-public operations op of the class* C *which have the property* $\{volatile = false\}$. [3]
2. ***Satisfaction of a set of constraints***
Let $Cons$ *be a set of OCL constraints.*
$I \models_c Cons$
if $I \models$ `inv` *for each invariant constraint* `inv` $\in Cons$ *and* $I \models_c$ `prepost` *for each pre- postcondition constraint* `prepost` $\in Cons$ *where* $c \in \{pc, tc, pec, tec\}$.

As a consequence of Definitions 1 and 2 we obtain the following result which can be proven by a straightforward propositional logic reasoning.

Theorem 1. *The transformation rules for pre- and postconditions presented in Section 4 are sound w.r.t. their corresponding satisfaction relations.*

[3] Note that also all inherited operations from superclasses of C are considered to be operations of C.

6.5 Semantics of Class Diagrams with Constraints

A class diagram $\mathcal{C}$ together with a set $Cons$ of constraints consisting of invariants attached to classes of $\mathcal{C}$ and pre- and postconditions attached to operations occurring in $\mathcal{C}$ can be represented by a pair $(\mathcal{C}, Cons)$. Its semantics is given by constraining the set of interpretation functions on $Opns(\mathcal{C})$ to those interpretations which satisfy the completion $Compl(Cons)$ of the given constraints (cf. Section 5 for the discussion of completions). Thereby, we distinguish again four possibilities according to the different satisfaction relations considered above.

$$Sem_c(\mathcal{C}, Cons) =$$
$$(State(\mathcal{C}), \{I : Opns(\mathcal{C}) \to StateTransFunct \mid I \models_c Compl(Cons)\})$$

where $c \in \{pc, tc, pec, tec\}$.

This definition induces in a straightforward way different kinds of correctness notions for implementations provided by object-oriented programs, written e.g. in Java. In fact, each program P intended to implement a given class diagram $\mathcal{C}$ with constraints $Cons$ can be considered as a particular interpretation function I_P. Then the program P is a correct implementation of $(\mathcal{C}, Cons)$ if $I_P \in Sem_c(\mathcal{C}, Cons)$.

7 Conclusion and Outlook

In this paper we have proposed several variants for a precise meaning to OCL constraints in the context of a UML model based on a formal abstract semantics. Moreover, we have defined a set of transformation rules to derive a single pair of pre- and postconditions from a system of independently specified constraints, taking into account inheritance as well as the semantic variants. To our knowledge, neither the comparative investigation of semantics nor the treatment of inheritance in the context of OCL constraints has been studied systematically up to now.

Which of the discussed variants is adequate for UML/OCL? Current UML documentation informally defines a semantics which is close to partial exception correctness. Nevertheless, other variants are used frequently in the literature, and with good reason. So the definitions used in the current standard may be too limited and special for all users of the language and all phases of the development process. This work is an attempt to provide the terminology and formal basis to clearly distinguish between the variants in a semantically richer variant of UML, as well as to build adequate tools including "bridges" between the variants.

References

1. Bidoit, M., Hennicker, R., Tort, F., Wirsing, M.: Correct Realizations of Interface Constraints with OCL. Proc. UML '99, The Unified Modeling Language - Beyond the Standard, Springer LNCS 1723, 399-415, 1999.

2. Brickford, M., Guaspari, D.: Lightweight Analysis of UML. Draft Technical Report, Odyssey Research Associates, 1998.
3. Cheesman, J., Daniels, J.: UML Components. Addison-Wesley, 2001
4. The CIP Language Group (F.L. Bauer et al.): The Munich Project CIP, Vol. I: The Wide Spectrum Language CIP-L. Springer LNCS 183, 1985
5. Demuth, B., Hussmann, H.: Using UML/OCL Constraints for Relational Database Design. Proc. UML '99, The Unified Modeling Language - Beyond the Standard, Springer LNCS 1723, 598-613, 1999.
6. Demuth, B., Hussmann, H., Loecher, St.: OCL as a Specification Language for Business Rules in Data Base Applications. Proc. UML '01, The Unified Modeling Language, To appear (Springer LNCS), 2001.
7. D'Souza, D.F., Wills, A.C.: Objects, Components and Frameworks with UML. The Catalysis Approach. Addison-Wesley, 1999
8. Fitzgerald, J., Larsen, P.G.: Modelling Systems. Practical Tools and Techniques in Software Development. Cambridge University Press, 1998
9. Floyd, R.W.: Assinging Meanings to Programs. *Proc. Symp. on Appl. Math.* **19**, American Mathematical Society 1967
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
11. Reus, B., Wirsing, M., Hennicker, R.: A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models. Proc. FASE 2001 - Fundamental Aspects of Software Engineering, Spirnger LNCS 2029, 285-300, 2001.
12. Hoare, C.A.R.: An Axiomatic Basis of Computer Programming. *Communications of the ACM* **12**, pp. 576-583, 1969
13. Hussmann, H., Demuth, B., Finger, F.: Modular Architecture for a Toolset Supporting OCL. Proc. UML 2000, The Unified Modeling Language - Advancing the Standard, Springer LNCS 1939, 278-293, 2000.
14. Liskov, B., Wing, J.: A Behavioral Notion of Subtyping. ACM Trans. on Prog. Lang. and Systems, Vol. 16 (6), 1811–1841, 1994
15. Meyer, B.: Object-oriented Software Construction. Prentice Hall, 1988.
16. Richters, M., Gogolla, M.: On Formalizing the UML Object Constraint Language OCL. Proc. 17th Int. Conf. Conceptual Modeling (ER '98) Springer LNCS 1507, 449-464, 1998.
17. Richters, M., Gogolla, M.: Validating UML Models and OCL Constraints. Proc. UML 2000, The Unified Modeling Language - Advancing the Standard, Springer LNCS 1939, 265-277, 2000.
18. OMG Unified Modeling Language Specification, Version 1.4 draft. February 2001
19. Warmer, J., Kleppe, A.: The Object Constraint Language. Precise Modeling with UML. Addison-Wesley, 1999
20. Klasse Objecten: Errata for "The Object Constraint Language, Precise Modeling with UML". Available at http://www.klasse.nl.