Definition and Correct Refinement of Operation Specifications

Thomas Baar, Slaviša Marković,
Frédéric Fondement, and Alfred Strohmeier

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
{thomas.baar, slavisa.markovic, frederic.fondement,
alfred.strohmeier}@epfl.ch

Abstract. Modern incremental and iterative software engineering processes advocate to build software systems by first creating a highly simplified and abstract model of the system which is then moved by applying a series of model improvements toward implementation. Models of software systems at any level of abstraction should contain, besides structural information, a precise description of the expected system behavior. This paper formalizes relations between models of the same system at different levels of abstraction, classifies approaches for describing behavior of system operations, and investigates how these system operation descriptions can be kept synchronized with frequent changes of the system's structure.

Keywords: Design by Contract, Refactorings, Refinements, System Operations, Graph Transformations, UML, OCL, QVT.

1 Introduction

In the Analysis phase of the software development lifecycle, the expected behavior of the system under development has to be described as clearly as possible. Many methodologies, e.g. Rational Unified Process [1], Catalysis [2], and Fusion [3], propose to start with a high-level class model that represents only coarsely the actual state space of the system. The system's behavior is modeled by operations attached to classes and precisely described by a contract consisting of a pre- and postcondition [4,5] (see also the survey [6] in this volume). In practice, contracts are often given only informally. Formal contracts – written in a formal specification language – have some obvious advantages such as being a non-ambiguous criterion for the correctness of the implemented system. After the Analysis phase has been finished, the developed class model serves as a starting point for further design activities. In the Design phase, the state space of the system is typically explored thoroughly in order to define the best possible way *how* the system can provide the behavior that has been specified in the Analysis phase. This includes to iteratively refine the current class model until it can be implemented directly in a chosen programming language.

Although the sketched development process can help to master the complexity of software development and to implement provably correct systems, it has not been widely adopted in industry yet. Many practitioners shy away from the effort to annotate class diagrams with formal contracts and to apply formal refinements when working on the system's design. We see two main reasons for the resistance to develop software by stepwise refinement. Firstly, the semantics of a contract language does not always meet the needs of developers. It is much more common to annotate a contract in an informal language such as English than in a formal language. Secondly, there is no common understanding on what refinement should mean and what it is good for in practice. Both problems are amplified by the lack of tool support. The contract language for UML, the Object Constraint Language (OCL), is not supported yet by any of the major UML tool vendors. A useful support for a contract language has to include also support for rewriting contracts when refining a (more) abstract class diagram to a (more) concrete one.

Despite their rare usage in practice, formal contract languages offer many advantages that fit well with recent trends in software engineering. Component-based development [7] is based on the idea of assembling applications from prefabricated modules (components). The functionality of a component is described best by a formal contract. Ideally, a component would carry also a formal proof to have implemented this contract correctly. Another motivation for using formal contract languages is model-driven development [8], which puts the modeling artifacts and not the implementation code to the forefront. The Model Driven Architecture (MDA) initiative of the OMG [9] aims at a framework for defining systems using a wide range of structural and behavioral views. The ultimate goal of MDA is to raise the level of abstraction at which systems are developed.

This paper discusses the purpose and semantics of contract languages and refinement steps. In Sect. 2, we divide existing formal contract languages into two groups called restrictive and constructive languages. After giving in Sect. 3 some formal definitions on the syntax and semantics of contracts, it is shown in Sect. 4 by example how a class diagram can be refined. Furthermore, the impact of a refinement on the syntactical correctness of operation contracts is shown. We make a proposal how refinement can be defined not as a syntactic transformation but based on the semantics of the involved class diagram. Based on this notion of refinement, we introduce correctness criteria for rewritten contracts in the refined diagram. For refactorings, which can be seen as a specific form of refinement, we discuss how operation contracts can be rewritten fully automatically without changing their semantics. In Sect. 5, constructive specification languages are discussed in more detail. Sometimes, constructive languages only allow to describe deterministic contracts. This shortcoming, however, can be remedied by integrating some elements of restrictive languages into constructive languages. Section 6 concludes the paper.

2 Restrictive vs. Constructive Languages

Formal languages for defining a contract, i.e. a pair of pre- and postcondition, can be divided into two groups. The distinction is based on the technique of formulating the postcondition of a contract. *Restrictive languages* focus in the postcondition on *Which properties must be satisfied in the post-state?* The postcondition is formalized as a predicate (a Boolean expression), which is evaluated to true in all valid post-states. Otherwise stated, the postcondition *restricts* the set of possible post-states. Well-known examples for restrictive languages are Eiffel [10], OCL [11], JML [12], and Z [13]. *Constructive languages*, instead, focus in the postcondition on *Which state transition is realized by the operation?* The contract prescribes how for a given pre-state the post-state is *constructed.* Well-known examples for constructive languages are Abstract State Machines (ASMs)[14], B [15], and UML's Action Language [16].

If an operation would be specified both with a restrictive and a constructive contract, then the properties of the post-state, which are given in the restrictive contract, could be entailed from a constructive contract. Some special purpose logics such as Hoare-Logic [17], Dynamic Logic [18] or even tools such as KeY [19] could be used to formally show this entailment relationship. The fact that restrictive contracts are comparably weak is also illustrated by the presence of the well-known *frame problem* [20,21]. Constructive languages, on the other hand, have the tendency to allow only deterministic specifications, which prevent any variations among the implementations of the specified operation.

We illustrate here briefly the main problems faced by restrictive and constructive specifications on a trivial example. A more detailed discussion on how the frame problem can be handled in restrictive languages is found in [22].

Let class `A` have two integer attributes `a1`, `a2` and one operation `op()`. The intended behavior of `op()` is to double the value of attribute `a1`. A contract in the restrictive language OCL typically looks as follows:

```
context A::op()
  post: self.a1 = self.a1@pre + self.a1@pre
```

Since this contract has no precondition the predicate true is implicitly assumed. The postcondition is given in form of a restriction on the post-state: a post-state is valid as long as the value of attribute `a1` on object `self` (which is the object on which `op()` is invoked) is doubled compared to the value of `a1` in the pre-state (represented by `a1@pre`). Although it was intended, the OCL contract does not imply that the change of attribute `a1` is the only effect of operation `op()`. According to this contract, also implementations of `op()` are correct that change the value of `a2`, or create new objects, or delete existing objects. A version of the OCL contract that really captures completely the intended behavior would be possible but its postcondition would be longwinded and had to mention explicitly all properties of the underlying class diagram that are not affected by `op()` (e.g. `self.a2=self.a2@pre and A.allInstances()=A.allInstances()@pre`).

The intended behavior could be given more easily and directly if a constructive language is used for formulating the contract. Operation `op()` could be specified in B by

$\mathrm{op}() \triangleq \mathrm{a1} := \mathrm{a1}*2$

Here, the precondition is omitted as well (implicitly true) and the postcondition is given in form of a pseudo-program which is 'executable' on a given pre-state. In difference to the restrictive contract, the semantics of the pseudo-program assumes an additional 'and nothing else changed' policy. Thus, implementations of `op()` that change, for instance, the value of attribute `a2` would be not correct according to this contract written in B.

As discussed more detailed in Sect. 5, constructive languages suffer from a problem that is opposite to the frame problem. Whereas restrictive contracts can hardly express which parts of the system state remain unchanged, constructive contracts can hardly expressed that some parts of the system can arbitrarily change.

3 A Formal Contract Language

This section formalizes a contract language, giving its syntax and semantics. We have chosen the language of UML class diagrams (see [23,24] for a general introduction) and the Object Constraint Language (OCL) [25,11] as a formal, restrictive language to specify contracts for operations. Our formalization will be the basis to define precisely our notion of refinement and correctness presented in Sect. 4.

The first two definitions formally capture the notion of class and object diagrams in a mathematical way. The well-known graphical notation of these diagrams are intentionally ignored here, but can be added straightforwardly as Fig. 1 illustrates.

Definition 1 (Class Diagram). *A class diagram cd is a tuple (*CLASS*,* ATT*,* ASSO*,* OPER*, owner, atttype, associates, mult, opsig,* $\preceq$*) where*

- CLASS*,* ATT*,* ASSO*,* OPER *are disjunctive finite sets containing symbols for classes, attributes, associations, and operations*
- *owner, atttype are total functions on* ATT *yielding the owning class and the type of attributes*
- *associates is a total function on* ASSO *yielding the list of classes connected by associations*
- *mult is a total function on* ASSO *yielding the list of multiplicities (sets of non-negative natural numbers) annotated on association ends*
- *opsig is a total function on* OPER *yielding the list of parameter types; we assume the owning class of the operation to be always the first element of the list*
- $\preceq$ *is a partial order on* CLASS *reflecting its generalization hierarchy*

The information given in a class diagram can easily be converted into a signature Σ of a sorted, first-order predicate logic. Every class name C becomes a sort symbol C, every attribute at becomes a function symbol $at : owner(at) \rightarrow atttype(at)$, the generalization hierarchy $\preceq$ is embedded in the sort hierarchy $\preceq$. Furthermore, we assume signature Σ to contain also entries for all declarations made in the standard OCL library. Thus, the set of sort symbols in Σ contains

OCL's standard types `Integer`, `Real`, `String`, `Set(`T`)`. There are function symbols in Σ for all pre-defined OCL operations such as `includes`: `Collection(`T`)` $\times\ T \rightarrow$ `Boolean` and the sort hierarchy $\preceq$ has entries for pre-defined types, e.g. `Integer` $\preceq$ `Real`, etc.

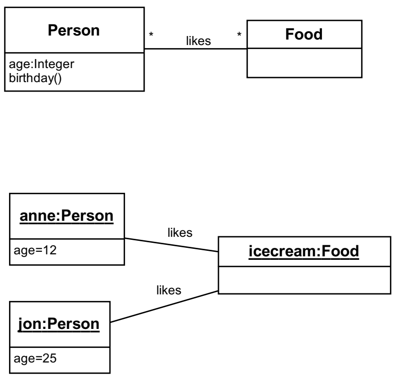
The first-order predicate logic for Σ is semantically interpreted by usual first-order structures. For the pre-defined symbols of the OCL library, a fixed interpretation is assumed. For example, the standard sort `Integer` is always interpreted as the set of natural numbers, the symbol `includes` is always interpreted as the set-theoretical *is-element-of* relationship, etc.

In our context, it is useful and common to call interpretations also *states*. That is the reason why the interpretations $\mathcal{I}$ of signature Σ are also denoted as $State_{\Sigma}$. Two interpretations for Σ can differ only in the interpretation of the symbols that stem from the class diagram. This part of interpretations can be depicted by *object diagrams*. Consequently, object diagrams and states are isomorphic structures. Object diagrams are denoted as a mathematical tuple in the same style as class diagrams. Figure 1 shows a simple example of a class and an object diagram both in the usual graphical notation and as mathematical tuples.

Definition 2 (Object Diagram). *Let cd=(*CLASS, ATT, ASSO, OPER*, owner, atttype, associates, mult, opsig,* $\preceq$*) be a given class diagram. An object diagram od for cd is a tuple (*OBJ, SLOT, LINK*) where*

– OBJ *is a total function on* CLASS. OBJ*(C) yields the finite set of all existing objects (we also say all instances) of class C.*
We also write OBJ_C *instead of* OBJ*(C).*
– SLOT *is a total function on* ATT. SLOT*(at) yields a total function from* OBJ*(owner(at)) to* OBJ*(atttype(at)).*
We also write SLOT_{at} *instead of* SLOT*(at).*
– LINK *is a total function on* ASSO. LINK*(as) yields a set of object lists where the i-th object in each list must be an instance of the i-th element of accociates(as).*
We also write LINK_{as} *instead of* LINK*(as).*

If as is a binary association (i.e. associates(as) is a list of length two), we use $opp_{as}(o)$ *as an abbreviation for* $\{o' \mid (o, o') \in Link_{as} \vee (o', o) \in Link_{as}\}$*. Otherwise stated,* $opp_{as}(o)$ *denotes the set of opposite ends of links in which object o is participating. If* $opp_{as}(o)$ *is a singleton set, we use* $opp_{as}(o)$ *also to refer to the contained element, i.e.* $opp_{as}(o)$ *might also stand as an abbreviation for* $\mu o' \mid (o, o') \in Link_{as} \vee (o', o) \in Link_{as}$*.*
– *The cardinality of* LINK_{as} *must satisfy the restrictions expressed by the multiplicities attached to as. More formally: Let as be an n-ary association and i=1,...,n. The function prj(i, list) extracts the i-th element from list. Then, for all i, for all tuple* $\in Link_{as}$ *the following holds:*
$\#\{tuple' \mid \bigwedge_{j=1..i-1,1+1,..,n} prj(j, tuple') = prj(j, tuple)\} \in prj(i, mult(as))$
– *If* $C1 \preceq C2$ *then* $Obj_{C1} \subseteq Obj_{C2}$*.*



$\textsc{Class} = \{\texttt{Person}, \texttt{Food}\}$,
$\textsc{Att} = \{\texttt{age}\}$, $\textsc{Asso} = \{\texttt{likes}\}$, $\textsc{Oper} = \{\texttt{birthday}\}$,
$owner = \{(\texttt{age}, \texttt{Person})\}$, $atttype = \{(\texttt{age}, \texttt{Integer})\}$,
$accociates = \{(\texttt{likes}, (\texttt{Person}, \texttt{Food}))\}$,
$mult = \{(\texttt{likes}, (\mathbb{N}^+, \mathbb{N}^+))\}$,
$opsig = \{(\texttt{birthday}, (\texttt{Person}))\}$,
$\preceq = \{\}$

$\textsc{Obj}_{\texttt{Person}} = \{\texttt{anne}, \texttt{jon}\}$, $\textsc{Obj}_{\texttt{Food}} = \{\texttt{icecream}\}$,
$\textsc{Slot}_{\texttt{age}} = \{(\texttt{anne}, 12), (\texttt{jon}, 25)\}$,
$\textsc{Link}_{\texttt{likes}} = \{(\texttt{anne}, \texttt{icecream}), (\texttt{jon}, \texttt{icecream})\}$

Fig. 1. Class and object diagram in both graphical and textual notation

A *contract* for an operation $op()$ is a pair $(pre, post)$ where pre, $post$ are predicate formulas over Σ. Both formulas can contain variables that are declared as formal parameters of $op()$ (note, that the pre-defined OCL variable *self* is handled in our formalism also as a formal parameter of $op()$). The precondition pre is evaluated on a given pre-state $s1$ and a given binding $argval$ of the formal parameters. The evaluation is defined formally by structural induction on all OCL expressions (see OCL's language definition [11] for details). We write $(s1, argval) \models_\Sigma pre$ iff pre is evaluated in $s1$ under the binding $argval$ to *true*. The postcondition $post$ is similarly evaluated in a pre-state $s1$, an argument binding $argval$, and a post-state $s2$. We write $(s1, argval, s2) \models_\Sigma post$ iff $post$ is evaluated to *true*.

Definition 3 (Semantics of Contract). *Let cd be a given class diagram, Σ the induced signature, $op()$ an operation in cd, and $(pre, post)$ a contract for $op()$.*

A label transition system *(LTS) for $op()$ is a pair $(State_\Sigma, \rho)$ where $State_\Sigma$ denotes all possible states for Σ and ρ is a subset of $State_\Sigma \times Argval \times State_\Sigma$. Here, Argval denotes all bindings of the formal parameters of $op()$ to concrete values.*

We say that $lts1 = (State_\Sigma, \rho_1)$ is larger *then $lts2 = (State_\Sigma, \rho_2)$, denoted by $lts2 \sqsubseteq lts1$, iff $\rho_2 \subseteq \rho_1$.*

The semantics of contract $(pre, post)$ is the largest LTS $sem_{op} = (State_\Sigma, \rho)$ such that $(s1, argval, s2) \in \rho$ implies that either $(s1, argval) \not\models_\Sigma pre$ or $(s1, argval, s2) \models_\Sigma post$.

Informally stated, the semantics of a contract is a LTS where the relation ρ contains exactly the state transitions which are possible according to the contract.

Definition 4 (Implementation, Partial/Total Correctness). *Let cd be a given class diagram, Σ the induced signature, $op()$ an operation in cd, and $(pre, post)$ a contract for $op()$.*

An implementation *of operation op() is a deterministic LTS* $(State_{\Sigma}, \rho)$ *on all possible states and argument bindings. A LTS* $(State_{\Sigma}, \rho)$ *is called* deterministic *on state s1 and argument binding argval iff* $(s1, argval, s2) \in \rho$ *and* $(s1, argval, s2') \in \rho$ *implies* $s2 = s2'$.

An implementation $(State_{\Sigma}, \rho)$ *of operation op() is called* partially correct *if* $(State, \rho) \sqsubseteq sem_{op}$.

An implementation $(State_{\Sigma}, \rho)$ *of operation op() is called* totally correct *if it is partially correct and* ρ *is total on all pre-states allowed by precondition pre. More formally: If* $(s1, argval) \models_{\Sigma} pre$ *then there exists* $s2$ *such that* $(s1, argval, s2) \in \rho$.

Definition 5 (Non-deterministic Contract). *Let (pre, post) be a contract for operation op(). We call this contract* deterministic *if* sem_{op} *is deterministic on all allowed states* $s1$ *and argument bindings argval that satisfy the precondition:* $(s1, argval) \models_{\Sigma} pre$*. Otherwise, the contract is called* non-deterministic*.*

Note that deterministic contracts allow only one implementation on the allowed pre-states.

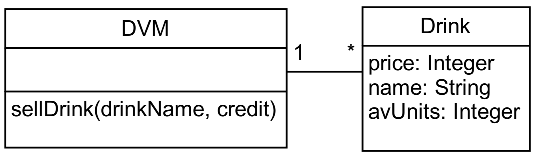
4 Correct Refinement of OCL Contracts

This section proposes a refinement notion that is purely based on the semantics of the involved class diagrams and does not impose any syntactical restrictions on them. Our approach is motivated by a simple case study on developing software to control a Drink Vending Machine (DVM). For some types of refinement, called *refactorings*, we derive in Sect. 4.4 some automatic rewriting rules for the contracts attached to the abstract class model.

4.1 Example: Drink Vending Machine (DVM)

A Drink Vending Machine (DVM) must be able to interact with both customers and service persons. The main functionality offered to customers is selling a drink. When a customer wants to buy a drink, s/he first has to select among the different drink kinds the machine offers, then to insert sufficient money, and finally to take the delivered drink from the drawer. A service person should be able to replenish the DVM with new drinks, to empty the money box of the DVM, to fix problems with the drawer, etc. For a realistic model of the DVM (see [26] for details), all possible exceptional cases have to be taken into account, e.g. that all drinks of a certain kind are sold out when a customer wants to buy it or that the capacity of the moneybox has been exceeded.

In this paper, we concentrate only on the formal specification of system operation `sellDrink()`. The operation `sellDrink()` is responsible for delivering drinks and for maintaining the number of available drinks in the DVM. Informally, this operation (1) checks, if the money inserted by the customer, called credit, is sufficient for buying the desired drink, (2) checks, if the desired drink is available, and (3) decrements the counter of how many drinks of the selected type are available in the system.



```
context DVM inv:
    self.drink->forAll(d1,d2| d1.name=d2.name implies d1=d2)
```

Fig. 2. System description on abstract layer

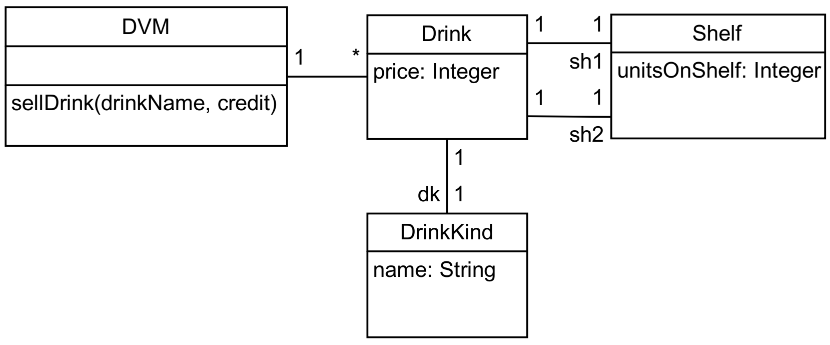
An initial model for the DVM could look like the class diagram given in Fig. 2. The operation `sellDrink()` declared on class `DVM` has two parameters. The parameter `drinkName` (of type `String`, what has been suppressed in the diagram) denotes the kind of drink the customer has selected, for instance tomato juice, orange juice, cola, beer, etc. The parameter `credit` (of type `Integer`) represents the amount of money inserted by the customer. The class `Drink` represents all possible kinds of drinks offered by the DVM. The attributes `name` and `price` are self-explaining, the attribute `avUnits` represents the number of units that are still available within the DVM for sale. The invariant attached to the class diagram ensures the uniqueness of drink names.

The intended behavior of `sellDrink` is deterministic. Whenever the selected kind of drink is available and the inserted money is sufficient to buy the drink, the value of attribute `avUnits` on the corresponding `Drink` object should be decremented by 1 and nothing else should happen. Be aware that at the beginning of a software development project, most operations are specified by deterministic contracts. This is due to the fact that the system model is quite abstract and hides most of the details that cause the complexity of the real system.

For the initial system model given in Fig. 2, a restrictive contract for operation `sellDrink()` would typically look like the following OCL specification:

```
context DVM::sellDrink(drinkName:String, credit:Integer)
pre:
    self.drink->select(d|d.name=drinkName and
                         d.price<=credit and
                         d.avUnits>0)->size() = 1
post:
    self.drink->select(d|d.name=drinkName)
                ->forAll(d1| d1.avUnits=d1.avUnits@pre-1)
```

The precondition formalizes the steps (1) and (2), which have been informally given above, and the postcondition decrements the attribute `avUnits` on the selected `Drink` object by one. This contract written in OCL, however, does not capture the intended behavior since it is not deterministic and would allow within the execution of `sellDrink()`, for example, to change the name of the drink, what is surely not intended. In other words, the semantics of the contract language does not fit the needs of developers working on analysis documents. Some restrictive languages have made provision to handle the frame problem by



```
context DVM inv:
    self.drink.dk->forAll(d1,d2| d1.name=d2.name implies d1=d2)
```

Fig. 3. System description on concrete layer

adding to the two standard clauses of a contract, pre- and postcondition, a third clause that is often called 'modifies'. The language OCL does not have such a third clause, yet. Nevertheless, we will use, despite all its weaknesses, the above shown contract for `sellDrink()` as the starting point for the next refinement of the system model.

4.2 Refinement of Class Diagrams

The initial system model shown in Fig. 2 has some obvious drawbacks, also known as *design smells*. The class `Drink` has three attributes, which all serve different purposes. The name of a drink is something very static whereas the price for one unit might be subject of very frequent changes. Furthermore, the number of units currently available is rather an attribute of the `DVM` than one of class `Drink`. Thus, all three attributes should be owned by different classes.

Let us assume that in addition to addressing these smells an improved model should take into account a new detail on DVMs, namely that for each kind of drinks a DVM has exactly two shelves. A new class diagram reflecting these changes is shown in Fig. 3. In the remainder of the paper, we name this diagram *cdiag* whereas the diagram in Fig. 2 is named *adiag* (for concrete and abstract diagram).

The improvements of the new diagram *cdiag* are basically two changes. Firstly, the attributes `name` and `price` were decoupled by (1) introducing a new class `DrinkKind`, (2) connecting it with class `Drink` using an association with multiplicities 1-1, and (3) moving attribute `name` from `Drink` to `DrinkKind`. Secondly, the new class `Shelf` became the owner of attribute `avUnits` (after renaming it to `unitsOnShelf`). Moreover, the new detail of the state space that each kind of drinks is stored on exactly two shelves is reflected by the two new associations `sh1` and `sh2`.

This informal definition how *adiag* has been refined to *cdiag* is not enough if one wants to argue formally on the correctness of such a step, e.g. in respect of the expected behavior of both original and refined system. Thus, we present now what refinement should mean in our context and how one can define the refinement relationship between two given (potentially completely different) diagrams in a formal way. Unlike the types of refinement foreseen in the UML (see [24]), our refinement definition does not map syntactical constructs from the concrete diagram to the abstract diagram. Instead, the main idea is to give a mapping from states of the concrete system to states of the abstract system. This technique goes back to a proposal made by Hoare in [27]. Please note that, unlike the refinement calculus presented in [28] for Eiffel, we do not aim here to bridge the gap from a UML model of a system to its implementation written in an implementation language such as Java.

Definition 6 (State Mapping). *Let cd^a, cd^c be two class diagrams, which are called* abstract *and* concrete *class diagram, respectively. A* refinement *is a relationship between them and is defined by a mapping function* map *for states and a mapping function* argmap *for argument vectors of operations.*

A state mapping is a total function $map : State_{\Sigma_{cd^c}} \to State_{\Sigma_{cd^a}}$. We will consider only such refinements for which a surjective mapping function map can be defined.

The function argmap is defined as a family of functions
$argmap_{op} : \mathcal{I}(opsig^c(op)) \to \mathcal{I}(opsig^a(op))$ *where* $op \in \textsc{Oper}$.

It is often possible to define for the same class diagrams cd^a, cd^c more than one mapping function. For example, let cd^a be the class diagram consisting of only one class B and two attributes b1, b2 of type Integer, and cd^c be the class diagram consisting of class B and two attributes b3, b4 of type Integer. Then, two possibilities of how *map* could be defined are:

1. $\textsc{Obj}^a_{\texttt{B}} := \textsc{Obj}^c_{\texttt{B}}$,
   $\textsc{Slot}^a_{\texttt{b1}}(o) := \textsc{Slot}^c_{\texttt{b3}}(o)$, $\textsc{Slot}^a_{\texttt{b2}}(o) := \textsc{Slot}^c_{\texttt{b4}}(o)$ where $o \in \textsc{Obj}^c_{\texttt{B}} = \textsc{Obj}^a_{\texttt{B}}$
   $\textsc{Link}^a := \textsc{Link}^c = \emptyset$
2. $\textsc{Obj}^a_{\texttt{B}} := \textsc{Obj}^c_{\texttt{B}}$,
   $\textsc{Slot}^a_{\texttt{b1}}(o) := \textsc{Slot}^c_{\texttt{b4}}(o)$, $\textsc{Slot}^a_{\texttt{b2}}(o) := \textsc{Slot}^c_{\texttt{b3}}(o)$ where $o \in \textsc{Obj}^c_{\texttt{B}} = \textsc{Obj}^a_{\texttt{B}}$
   $\textsc{Link}^a := \textsc{Link}^c = \emptyset$

In its first version, *map* assigns the value for attribute b3, b4 to the values for the abstract attributes b1, b2. In the second version, it is vice versa, b3 is mapped to b2 and b4 is mapped to b1. For the sake of brevity, we often omit in the definition of *map* the assignment for those components in the abstract object diagram that remain the same as in the concrete object diagram. For example, the first refinement could also be given as $\textsc{Slot}^a_{\texttt{b1}}(o) := \textsc{Slot}^c_{\texttt{b3}}(o), \textsc{Slot}^a_{\texttt{b2}}(o) := \textsc{Slot}^c_{\texttt{b4}}(o)$ since $\textsc{Obj}^a_{\texttt{B}}, \textsc{Link}^a$ are the same as $\textsc{Obj}^c_{\texttt{B}}, \textsc{Link}^c$. An even shorter definition would be possible: $\textsc{Slot}^a_{\texttt{b1}} := \textsc{Slot}^c_{\texttt{b3}}, \textsc{Slot}^a_{\texttt{b2}} := \textsc{Slot}^c_{\texttt{b4}}$.

The second mapping function *argmap* maps the input parameters for the operation of the concrete layer to input parameters for the operation of the abstract layer. Suppose, class `B` had an operation `inc(Integer)` in both cd^a and cd^c. The mapping function $argmap_{\texttt{op}}$ could be defined as follows:

$argmap_{\texttt{op}}((o,i)) = (o, i+3)$

The variable o represents the object on which `op` is invoked. An invocation with parameter i at the concrete layer is mapped to an invocation on the same object o with parameter $i+3$ at the abstract layer. As for *map*, we will omit the definition for *argmap* if it maps argument vectors of the concrete layer to identical vectors on the abstract layer.

The intended mapping for the DVM example is:

$\text{SLOT}^a_{\texttt{name}}(opp_{\texttt{dk}}(o)) = \text{SLOT}^c_{\texttt{name}}(o)$
$\text{SLOT}^a_{\texttt{avUnits}}(opp_{\texttt{sh1}}(o)) = \text{SLOT}^c_{\texttt{unitsOnShelf}}(o) + \text{SLOT}^c_{\texttt{unitsOnShelf}}(opp_{\texttt{sh2}}(opp_{\texttt{sh1}}(o)))$

4.3 Refinement of Contracts

It is obvious that the formal contract for `sellDrink()` in *adiag* cannot be simply copied to *cdiag*. The copied contract would be neither syntactically nor semantically correct. Contracts have to be adapted to the new class diagram, a new version for `sellDrink()` could look like the following:

```
context DVM::sellDrink(drinkName:String, credit:Integer)
pre:
    self.drink->select(d|d.dk.name=drinkName and
                          d.price<=credit and
                         (d.sh1.unitsOnShelf>0 or
                          d.sh2.unitsOnShelf>0))->size() = 1

post:
    self.drink->select(d|d.dk.name=drinkName)
      ->forAll(d1|
        d1.sh1.unitsOnShelf+d1.sh2.unitsOnShelf=
        d1.sh1.unitsOnShelf@pre+d1.sh2.unitsOnShelf@pre-1)
```

For the precondition, the attribute access `d.name` in the original contract was changed to `d.dk.name` in order to reflect moving of attribute `name` from `Drink` to `DrinkKind`. Furthermore, it must be tested now if at least one shelf has units available. More interesting changes have been made in the postcondition. The original postcondition, which strived to describe a deterministic state change, now became intentionally non-deterministic. Instead of decreasing attribute `avUnits`, an implementation of `sellDrink()` could change attribute `unitsOnShelf` either for the first or the second shelf (`sh1`, `sh2`). This is achieved by the *under-specified* postcondition saying that the sum of `unitsOnShelf` for both shelves is decreased by one.

Note that the new contract leaves the decision open *which one* of the shelves decreases its number of units. In other words, an implementation of `sellDrink()` that first makes the first shelf empty before selling drinks from the second shelf should be possible as well as an implementation that sells units from the second shelf before the ones from the first shelf or an implementation that alternates between both shelves. Of course, an implementation has to realize a concrete, fixed algorithm but the decision which algorithm to take is deferred here to a later phase of the software development project.

In the following, we want to answer the question whether the new version of the OCL contract is correct in respect to the contract given for `sellDrink` in *adiag*. A very basic criterion for correctness of the refined system is that every state transition which the concrete system is allowed to make has its 'counterpart' in the abstract system. Otherwise stated, whenever one can observe a behavior on the concrete system, 'the corresponding behavior' on the abstract system is allowed as well. The, somehow, imprecise terms 'counterpart' and 'corresponding behavior' are made clear in the following formal definition as a projection of behavior from the concrete layer to the abstract layer under the state mapping, which is defined when the underlying class diagram is refined.

Definition 7 (Correct Contract Refinement). *Let class diagrams* cd^a,cd^c *and refinement functions map, argmap be given. Furthermore, let op() be an operation declared in both diagrams, and* $(pre^a, post^a)$, $(pre^c, post^c)$ *be its contracts. The semantics of the contracts are the LTSs* $sem^a_{op} = (State_{\Sigma_{cd^a}}, \rho^a)$, $sem^c_{op} = (State_{\Sigma_{cd^c}}, \rho^c)$.

The contract $(pre^c, post^c)$ *for op() is called to be a* correct refinement *of the abstract contract* $(pre^a, post^a)$ *if the following holds:*

For all argument bindings argval for op() at the concrete layer, for all $s1, s2 \in State_{\Sigma_{cd^c}}$*: if* $(s1, argval, s2) \in \rho^c$ *then* $(map(s1), argmap_{op}(argval), map(s2)) \in \rho^a$

After giving a formal correctness criterion for the refinement of contracts, it is, of course, interesting to discuss whether or not the refined contract for `sellDrink()` shown above is correct according to this criterion. We do not answer this question immediately but will describe in the next subsection a technique to ensure the correctness of simple refinements. The same technique is powerful enough also to argue on the correctness of more complicated refinements.

4.4 Refactorings as Simple Refinements

The refinement from *adiag* to *cdiag* was defined so far as a monolithic step. One could also think to achieve the same by a concatenation of much smaller refinements: 1) create class `DrinkKind` and connect it to `Drink` with 1-1 association 2) move attribute `name` from `Drink` to `DrinkKind` 3) create class `Shelf` and connect it to `Drink` with 1-1 association 4) move attribute `avUnits` from `Drink` to `Shelf` 5) rename attribute `avUnits` as `unitsOnShelf` 6) add a second 1-1 association between `Drink` and `Shelf`.

From these six steps, the first five steps do not change (up to isomorphism) the state space of the system but just restructure the model. Such steps, called *refactorings*, are small improvements that lead to better design since they directly address poor structures of the model (smells). A smell can be the duplication of attributes or operations, a heavy class having too many responsibilities, too many dependencies between classes, etc. Typical changes done by refactorings include moving attributes and operations up and down in the inheritance hierarchy of the classes, moving attributes and operations to newly created classes, giving attributes, operations, classes a new name, etc. The main characteristic of these changes is that they do not make the system description closer to the implementation level but keep it at the same level of abstraction.

In recent years, much research has been devoted to refactoring (see [29] for an overview). Most of these works, however, have concentrated on the refactoring of implementation code. Fowler gives in [30] a catalog of refactorings for Java programs. A typical refactoring rule describes in a first step changes on the Java declarations, e.g. moving a field to a new class to make the original class less heavy, and in a second step, how the remaining Java program must be updated in order to become consistent with the changed declaration, e.g. every access to the original field must be forwarded to the new class.

In [31], we have formalized a catalog of basic refactorings for UML class diagrams together with the necessary changes on attached OCL constraints. The above given correctness criterion offers now the possibility to argue on the semantical correctness of the refactorings. To do this, we have to assume that every refactoring is associated with a unique mapping function. This mapping function is not part of refactoring catalogs yet, but is in most cases obvious.

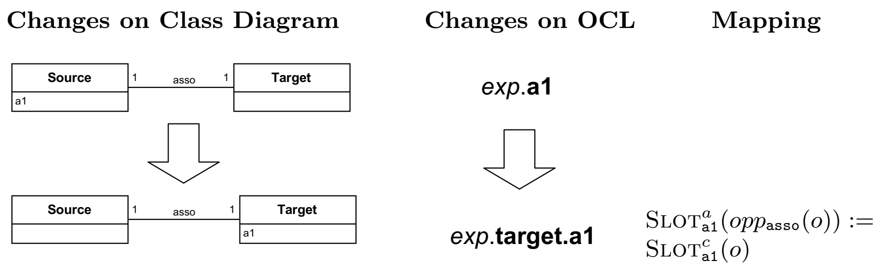


Fig. 4. Refactoring MoveAttribute

As an example we formalize in Fig. 4 the refactoring *MoveAttribute*, which moves in a first step the attribute from the source class to a target class if both classes are connected by an 1-1 association. Actually, there are even more side conditions that must hold, e.g. that the target class has not already an attribute with the same name as the moved attribute. These side conditions are dropped here for the sake of brevity, the interested reader is referred to the formalized version of this rule given in [31]. In a second step, all OCL constraints attached to the class model must be updated by substituting each attribute access of form *exp*`.a1` with the new expression *exp*`.target.a1`. The expression *exp*`.target` is a navigation from source to target class. Note that updating the attached OCL constraints can be done automatically.

We finish this section with a theorem on the correctness of the defined refactoring.

Theorem 1 (Correctness of MoveAttribute). *The refactoring MoveAttribute refines contracts correctly.*

Proof. Let cd^a, cd^c be the original and refactored diagram, $op()$ an operation, and $(pre^a, post^a)$, $(pre^c, post^c)$ the original and the refactored contract for $op()$. The two contracts induce the two LTSs $sem^a_{op} = (State^a_\Sigma, \rho^a)$ and $sem^c_{op} = (State^c_\Sigma, \rho^c)$. According to the correctness criterion we have to show that for every tuple $(s1, argval, s2) \in \rho^c$, there is a tuple $(map(s1), argmap(argval), map(s2)) \in \rho^a$.
The condition $(s1, argval, s2) \in \rho^c$ means $(s1, argval) \models_{\Sigma_{cd^c}} pre^c$ and $(s1, argval, s2) \models_{\Sigma_{cd^c}} post^c$. The constraints pre^c, $post^c$ are by construction only different from pre^a, $post^a$ at subexpressions of form *exp*`.target.a1`. According to the semantics of OCL, this expression is evaluated in s to the same value as *exp*`.a1` in $map(s)$. Thus, we have
$(s1, argval) \models_{\Sigma_{cd^c}} pre^c$ if and only if $(map(s1), map(argval)) \models_{\Sigma_{cd^a}} pre^a$ and, furthermore, $(s1, argval, s2) \models_{\Sigma_{cd^c}} post^c$ if and only if $(map(s1), argmap(argval), map(s2)) \models_{\Sigma_{cd^a}} post^a$. □

5 Constructive Specifications

In the preceding section, we have discussed the problems related with refinement of class diagrams and how contracts have to be updated accordingly. We have seen, that a refinement sometimes requires to rewrite a deterministic contract by a non-deterministic one.

For the DVM example we tried to capture the intended behavior of operation `sellDrink()` with a restrictive constraint in OCL. Due to the immanent frame problem of restrictive languages, it is practically impossible to formalize deterministic contracts, which, however, are often needed in the first phases of a software development project.

In this section, we discuss how more appropriate contracts for `sellDrink()` can be given using a constructive specification language. The specification language of our choice is QVT [32], a special form of graph transformations.

5.1 Graph Transformations

Graph transformations (see [33] for an overview) were originally developed to manage the manipulation of graphs. Since system states are easily representable as graphs, they can also be used as a tool to describe state changes, i.e. the intended behavior of operations.

A graph is manipulated by applying a graph transformation rule on it. Every graph transformation rule consists of a Left Hand Side (LHS) pattern and a Right Hand Side (RHS) pattern. Both patterns consist of labeled nodes and links, which might occur in both patterns. The rule is applied by, firstly, searching subgraphs in the given graph that match with LHS and, secondly, by rewriting the matching subgraphs with new subgraphs derived from RHS. If a node/link in the given graph matches with a node/link that occurs, according to its label, only in LHS then this node/link is removed from the given graph. If a node/link occurs only in RHS then a corresponding element is created. Nodes can also have slots for attribute values. These values are updated according to the allowed values of variables occurring in the rule. Finally, a rule has a name (reflecting the operation it specifies) and parameters (reflecting the signature of the operation).

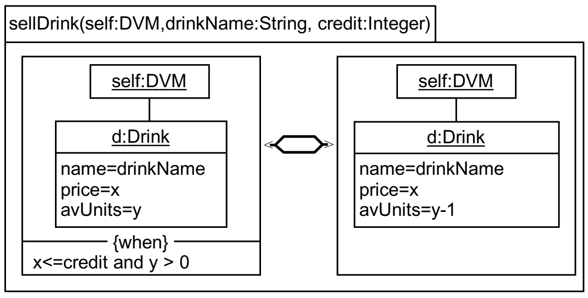


Fig. 5. Constructive contract for `sellDrink()` on abstract layer

As an example, we consider the specification of `sellDrink()` as shown in Fig. 5. This rule has to be read – as an operation contract – as follows: Whenever in a pre-state an object `d` of class `Drink` is linked with the object `self` on which `sellDrink()` was invoked and the value of its attribute `name` matches with parameter `drinkName` and the value for attribute `price` is less than or equal to parameter `credit` and the value of attribute `avUnits` is greater than 0, then the post-state is derived from the pre-state by decreasing the value of attribute `avUnits` by 1.

Note that the post-state is *constructed* from the pre-state. This is despite the fact that the specification is, unlike in B or ASM, not given in form of a pseudo-program but by a pair of matching patterns. The given contract is truly deterministic since it completely prescribes the update from the pre- to the post-state. The semantics of the graph transformation rules stipulates that all parts of the pre-state that do not match with LHS remain unchanged.

5.2 Graph Transformations for Non-deterministic Specifications

One common problem of constructive languages is the tendency to allow only the formulation of deterministic contracts (due to the construction of a unique post-state). Actually, many graph transformation systems, e.g. AGG [34], allow only to describe deterministic rules because they insist on having an executable specification. The formalism of our choice, QVT, is an exception and allows to use variables that only occur in RHS. Consequently, these variables are not bound after the first step of the rule application (the matching of a subgraph with LHS). The value for the variables can freely be chosen in the second rule application step, when the matching subgraph is rewritten with a new graph derived from RHS. The variable values, however, can be restricted by an OCL constraint given in the when-clause of the rule.

The non-deterministic contract for `sellDrink` on the concrete layer is shown in Fig. 6. There are two new variables y1' and y2' in RHS whose values can be chosen non-deterministically as long as the restrictions imposed by the OCL constraint in the when-clause are obeyed.

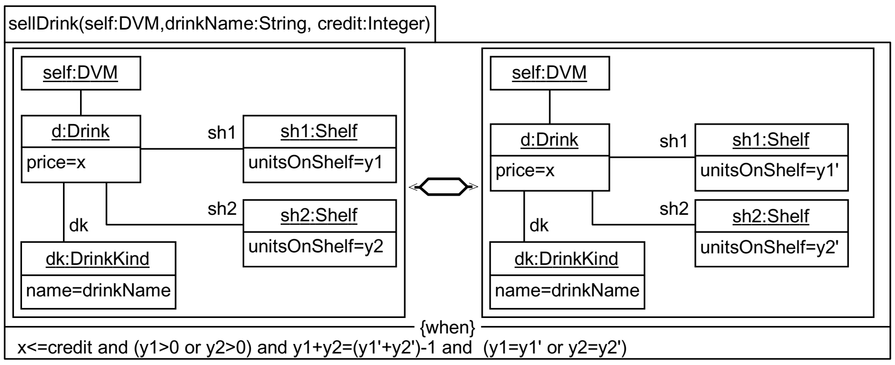


Fig. 6. Constructive, non-deterministic contract for `sellDrink` on concrete layer

The integration of a when-clause into transformation rules can be seen as an attempt to mix constructive with restrictive specification style. This idea is actually not new, also the constructive language B provides with ANY-WHERE and the language ASM with its non-deterministic choices analogous constructs. These non-deterministic constructs in turn have again inspired the language designers of restrictive languages to include them. For instance offers OCL a construct any() that should mimic the ANY-WHERE construct of B. The integration of any() in OCL has caused, however, a lot of contradictions in the language semantics as analyzed in [35]. This is another striking example for a mis-conception, if the fundamental differences between restrictive and constructive specifications languages are not sufficiently understood.

6 Conclusions

In this paper, we have formalized relations between models of the same software system situated at different levels of abstraction. We assume the system to be described by UML class diagrams with OCL constraints attached, but our results can easily be applied also to other specification formalisms.

Moreover, we have given a classification of formal contract definition languages in respect to the underlying specification technique they offer. For graph transformations, which can be seen as a constructive specification language, we propose an approach to express non-determinism by enriching them with restrictive specification elements.

Another contribution of this paper is the investigation how changes made on structural part of a model can influence contracts for operations. In order to cope with this problem, we have defined criteria for the correctness of contract refinements. We were able to prove for a simple kind of standard refinement, a well-known refactoring rule, that its application preserves the semantical correctness of contracts.

Acknowledgements

This work was supported by HASLER-Foundation, project DICS-1850 and by Swiss National Science Foundation, grant 200020-101596.

References

1. Phillippe Kruchten. *The Rational Unified Process.* Object Technology Series. Addison-Wesley, 1999.
2. Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Object Technology Series. Addison-Wesley, 1998.
3. Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.
4. Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.
5. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, second edition, 1997.
6. Bertrand Meyer. Dependable software. In Jürg Kohlas, Bertrand Meyer, and André Schiper, editors, *Dependable Systems: Software, Computing, Networks*, volume 4028 of *LNCS*. Springer, 2006. (this volume).
7. Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1997.
8. Stuart Kent. Model driven engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *Proceedings of Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.
9. Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled.* Addison-Wesley, 2004.
10. Bertrand Meyer. *Eiffel – The Language.* Prentice-Hall, Englewood Cliffs, 1992.
11. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
12. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for java. Technical Report TR 98-06-rev28, Department of Computer Science, Iowa State University, 2005. Last revision July 2005, available from www.jmlspecs.org.
13. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1992.
14. Egon Börger and Robert Stärk. *Abstract State Machiness.* Springer, 2003.
15. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, August 1996.
16. OMG. UML 1.5 Specification. OMG Document formal/03-03-01, March 2003.

17. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
18. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000.
19. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.
20. J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, pages 463–502, 1969.
21. A. Borgida, J. Mylopolous, and R. Reiter. ...And Nothing Else Changes: The Frame Problem in Procedure Specifications. In *Proceedings of ICSE-15*, pages 303–314. IEEE Computer Society Press, 1993.
22. Thomas Baar. OCL and graph transformations – a symbiotic alliance to alleviate the frame problem. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *LNCS*, pages 20–31. Springer, 2006.
23. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, second edition, 2005.
24. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, second edition, 2005.
25. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.
26. Alfred Strohmeier, Thomas Baar, and Shane Sendall. Applying Fondue to specify a drink vending machine. *Electronic Notes in Theoretical Computer Science, Proceedings of OCL 2.0 Workshop at UML'03*, 102:155–173, 2004.
27. C. A. R. Hoare. Proof of correctness of data representation. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *LNCS*, pages 183–193. Springer, 1975.
28. Richard F. Paige and Jonathan S. Ostroff. ERC – an object-oriented refinement calculus for Eiffel. *Formal Aspects of Computing*, 16:51–79, April 2004.
29. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.
30. Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
31. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In Lionel Briand and Clay Williams, editors, *Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
32. OMG. Revised submission for MOF 2.0, Query/Views/Transformations, version 1.8. OMG Document ad/04-10-11, Dec 2004.
33. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
34. AGG team. AGG homepage. http: //tfs.cs.tu-berlin.de/agg, 2005.
35. Thomas Baar. Non-deterministic constructs in OCL – what does any() mean. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *Proc. 12th SDL Forum, Grimstad, Norway, June 2005*, volume 3530 of *LNCS*, pages 32–46. Springer, 2005.