Non-deterministic Constructs in OCL – What does any() Mean

Thomas Baar*

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, Lausanne, Switzerland
Email: thomas.baar@epfl.ch

Abstract. The Object Constraint Language (OCL) offers so-called non-deterministic constructs which are often only poorly understood even by OCL experts. They are widely ignored in the OCL literature, their semantics given in the official language description of OCL is ill-defined, and none of today's OCL tools support them in a consistent way.
The source of the poor understanding and ill-defined semantics is, as identified in this paper, OCL's attempt to adopt the concept of non-determinism from other specification languages with fundamentally different semantical foundations. While this insight helps to improve the understanding of non-deterministic constructs it also shows that there are some formidable obstacles for their integration into OCL.
However, in some cases, non-deterministic constructs can be read as abbreviations for more complex deterministic constructs and can help to formulate a specification in a more understandable way. Thus, we suggest to integrate non-deterministic constructs in other specification languages such as Z, JML, Eiffel whose semantical foundations are similar to those of OCL.

1 Introduction

Specification languages describe properties of systems on a certain level of abstraction. System development typically requires a broad spectrum of specification languages which must be able to cope with different properties (structural, behavioral, non-functional) in different stages of development. This was the main motivation in the early 90-ies to tightly bind 7 different diagrammatic languages to the Unified Modeling Language (UML)[1].

The UML language description [2, 3] defines the integrated languages and their interconnections in terms of a meta-model that is written in MOF (a derivate of UML class diagrams) and the Object Constraint Language (OCL). The meta-modeling technique has become extremely popular in recent years and is used more and more often to define other specification and even programming languages.

This development has promoted the use of OCL through the fact that well-formedness rules of the syntax in meta-model based language definitions are described by OCL constraints. Since well-formedness rules are the core of a

* This work was partially supported by Hasler-Foundation, project DICS-1850.

language description, the application of OCL in meta-models requires an exceptionally deep understanding of this constraint language. Mistakes, made during the definition of a new language, will obfuscate its syntax and also its semantics and thus the purpose of the new language itself.

Widely neglected and often misunderstood are up to now so-called *non-deterministic constructs* in OCL. The most basic non-deterministic construct is the library operation `asSequence()` that expects as an argument a term of type `Set(`T`)`[1] and yields a term of type `Sequence(`T`)`. Semantically, `asSequence()` is used to turn a set into a sequence that has the same elements as the set. The construct `asSequence()` is called non-deterministic, because it imposes a non-deterministically chosen ordering on the elements in the resulting sequence which is not given for the elements of the argument set. As a second non-deterministic construct, the operation `any()` is offered by the OCL library. It expects a term of type `Set(`T`)` and yields a term of type T. Semantically, the operation `any()` can be used to select non-deterministically an element from a set. The non-deterministic selection could be simulated by turning the set into a sequence imposing an ordering on its elements and, in a second step, by taking that element which has order number 1. For this reason, `any()` can be seen as an abbreviation for `asSequence()` concatenated with `first()`, another library operation which yields the first element of a sequence if the sequence has at least one element and *undef*, otherwise.

As it is shown in section 2, there are some formidable obstacles for defining a formal semantics for non-deterministic constructs in OCL. The main argument goes as follows: The semantics of constraints attached to a system description is defined on the basis of constraint evaluations in concrete system states. For instance, a constraint attached as an invariant to the system description characterizes the allowed system states for which the constraint must be evaluated to *true*. This simple semantics, however, cannot be applied to an invariant containing non-deterministic constructs because the evaluation of the invariant in a given state might yield more than one result, for example, *true* and *false*.

The problematic semantics of non-deterministic constructs in OCL makes users understandably reluctant to take advantage of non-determinism. For example, the UML metamodel [2, 3] (both documents have together 839 pages) is authored by some of the leading experts for UML, but `any()` is the only non-deterministic construct that occurs (21 times). Even more interesting, the construct `any()` is always applied on sets containing exactly one element. When applied on a singleton set, however, the construct `any()` can be seen as a deterministic operation. Thus, the whole UML metamodel contains not a single, truly non-deterministic constraint.

Although `asSequence()` is the more basic construct compared to `any()` this paper concentrates on the semantics of `any()` for two reasons. Firstly, the non-determinism introduced by `asSequence()` cannot be captured by an evaluation based semantics without losing other important logical properties. Secondly, the

[1] `Set(`T`)` is a parameterized type where T is a placeholder for subtypes of the predefined type `OclAny`.

only non-deterministic construct used in practice, and this also only very rarely, is `any()` – that is, the combination of `asSequence()` and `first()`. Fortunately, a constraint using `any()` can, as we will see, often be rephrased by another constraint that has the same 'intended' meaning, but only contains deterministic constructs.

For the design of the specification language OCL, our results have two consequences. In principle, the evaluation based semantics prevents OCL having non-deterministic constructs. Thus, we propose to delete `asSequence()` from the OCL library. The construct `any()` can remain in the library with the same meaning it currently has (non-deterministic selection of one element from a set) but not as an abbreviation for `asSequence()->first()`. Instead, `any()` should be introduced as an abbreviation according to the transformation algorithm given in section 4.

The remainder of the paper is structured as follows. Section 2 points out the problems in the current semantics of OCL caused by non-deterministic constructs. A subsection illustrates how the unsolved problems have a disastrous impact on the tool support for OCL. Section 3 compares OCL with other specification languages and identify the reasons why OCL tries to offer non-deterministic constructs. This comparison will clarify what the *intended meaning* of the construct `any()` is. After the role, the construct `any()` plays in OCL, is understood, Section 4 presents two attempts to capture the intended meaning formally. Both approaches have some limits, but the limitations of the second approach based on transformation are irrelevant for practical specifications. Section 5 concludes the paper.

2 Problems with the any()-Construct in OCL

The Object Constraint Language (OCL), specified in its most recent version 2.0 in [4], is a strongly typed, term-based specification language. Terms are either atomic, for example variables, or are composed of an operation that is applied to subterms. Terms of the predefined type `Boolean` are called *constraints*.

When attached to a class diagram, the purpose of an OCL constraint is to *restrict* the allowed states of the system described by the class diagram. If a constraint is attached as an invariant, then the state of the system must always *conform* to that constraint. If a constraint is attached as a pre- or post-condition of a system operation, then the system state must conform to the constraint whenever the operation is invoked or has terminated.

The meaning (semantics) of an OCL constraint must clarify which of the possible system states conform to it and which of them do not conform. The separation between conforming and non-conforming states is implicitly given by an *evaluation function eval* that yields, applied on a concretely given state and a constraint, one of OCL's three truth-values *true*, *false*, *undef*. The function *eval* is defined in [4] by structural induction on all OCL terms.

The application of *eval* on a constraint *constr* and a state *st* is called *evaluation of constr in st*. The state *st* conforms to *constr* if and only if *constr* is evaluated in *st* to *true*. If a constraint is attached as a post-condition to the sys-

tem and contains the `@pre` operator, then its evaluation is analogously defined on a pair of states instead of a single state.

In the OCL language description [4], `any()` is declared as an operation with one argument[2] of type `Collection(`T`)` and return type T. More precisely, the operation `any()` is used in composed terms of the form *src*`->any()`, where *src* has the type `Collection(`T`)` and the composed term is of type T. Most often, `any()` is applied to terms of type `Set(`T`)` (a subtype of `Collection(`T`)`) and, to facilitate our argument, we will assume in the rest of the paper *src* to be of type `Set(`T`)`.

The evaluation of terms of form *src*`->any()` is described in the OCL language specification as a non-deterministic choice from the set that is obtained by evaluation of *src* (see [4, page A-19]). If the evaluation of *src* yields an empty set or a singleton set, the evaluation of *src*`->any()` yields *undef* or the single element of the singleton respectively. In these two exceptional cases, the evaluation of *src*`->any()` is deterministic and well-defined. In all other cases, the non-deterministic evaluation can cause serious problems as a first example illustrates:

context `Foo :: foo1():Integer`
post: `result = Set{1,2}->any()`

The term `Set{1,2}->any()` is non-deterministically evaluated in any state to 1 or 2. The official OCL semantics in [4] does not clarify the consequences of non-deterministic evaluation for the conformance of states to a non-deterministic constraint. Suppose, the system operation `foo1()` terminates in a state *st* and returns[3] for example the value 1. If `Set{1,2}->any()` is evaluated to 1, then *st* would conform to the post-condition but does the same state conform if 2 is non-deterministically chosen by the evaluation algorithm instead of 1? It seems the only thing that can be concluded from the OCL semantics, is, that all post-states in which `foo1()` returns a value different from 1 and 2 do not conform to the post-condition. It remains an open question if this indeed completely captures the meaning of that constraint.

The next example is a slight variation of the last one.

context `Foo :: foo2():Integer`
post: **if** `Set{1,2}->any() = Set{1,2}->any()`
then `result = 1`
else `result = 2`
endif

Would this specification allow a post-state where `foo2()` returns 1? One could argue 'yes', because it is possible to find among all non-deterministic evaluations for both `any()`-terms such an evaluation where the if-condition is evaluated to *true*.

[2] Sometimes, `any()` is used with a second argument of type `Boolean` that serves as a guard. Note, that terms of the form *src*`->any(`*guard*`)` can be rewritten to *src*`->select(`*guard*`)->any()`.

[3] The return value of an operation is represented in post-conditions by the predefined variable `result`.

Analogously, one could argue for conformance of a post-state with return value 2, because an evaluation could be found where the if-condition is evaluated to *false*. This would require that the two `any()`-terms are evaluated differently, for instance the first to 1 and the second to 2.

A conformant state with return value 2, however, would contradict the fundamental logical law that equality is a reflexive relation. Note, that the if-condition is of form X `=` X and most logics allow to simplify this to *true*. Consequently, the `if-then-else` expression would collapse to `result = 1`, which would not allow 2 as a return value.

2.1 Current Tool Support for any()

Current tools for OCL (see [5] for an overview) have either not implemented the `any()` construct (a sign that non-deterministic constructs are not well-understood yet) or have implemented it in a way which contradicts basic and widely accepted laws in logic.

For instance, as one of the few tools that can handle `any()`, OCLE [6] evaluates the expression `Set{1,2}->any() = Set{1,2}->any()` always (!) to *true* whereas `Set{1,2}->any() = Set{2,1}->any()` is always evaluated to *false*. This contradicts the law that for a set the ordering of the elements is not important; the term `Set{1,2}` should denote the same set as `Set{2,1}`.

Probably, the authors of OCLE have understood the non-determinism of the evaluation function in OCL in such a way, that the decision, which among all possible evaluations should be chosen, can be made by the tool. But such a setting would give one tool the freedom to confirm the conformance of a state to a constraint while another tool comes to the opposite conclusion for exactly the same state and the same constraint. Finally, the meaning of an OCL constraint (the decision which of the system states conform to it), could depend entirely on the tool that is used to process that constraint!

3 Non-determinism versus Under-specification

In order to understand the construct `any()` offered by OCL it is helpful to concentrate on the usage of OCL as a contract specification language. A contract [7] for a system operation describes its behavior in terms of pre- and post-conditions.

3.1 Constructive versus Restrictive Languages

Contract specification languages can be classified into two groups. The classification is based on the technique in which post-conditions are formulated (the formulation of pre-conditions is much more uniform than for post-conditions and relies always on a dialect of predicate logic).

Languages belonging to the first group, *constructive specification languages*, provide pseudo-code for the formulation of the post-condition. The pseudo-code allows specification of the operation's behavior in the form of an algorithm. In other words, the transition of the system from the pre-state to the post-state is given by the sequential, conditional (and sometimes also parallel) composition of more atomic state-transitions. The pseudo-code often resembles imperative pro-

gramming languages with their basic control structures (assignment, sequential and parallel execution, if-then-else, loops). Two of the most prominent examples of constructive specification languages are Abstract State Machines (ASM) and B. The specification given in the post-condition is called *update* in the ASM terminology and *generalized substitution* in B.

Languages of the second group, *restrictive specification languages*, offer for the formulation of the post-condition basically the same formalism as for the pre-condition. In such languages, a post-condition restricts the set of possible post-states. The intention is not to describe *how* the post-state is 'constructed' from the pre-state (even if this is possible in some situations as our examples will show). Nevertheless, it is possible to specify in the post-condition how the post-state is related to the pre-state. For that reason, restrictions can be formulated on the value of the state variables in the post-state as well as in the pre-state because all such languages allow the post-condition to refer to both pre- and post-state. For example, in OCL, `att1 > att1@pre` means that the value of `att1` in the post-state must be greater than its value in the pre-state.

Well-known examples for restrictive specification languages are Hoare-Triple, Dynamic Logic, Eiffel, Java Modeling Language (JML), and Z.

Non-deterministic constructs play an important role in constructive languages, but they cannot, as seen in the last section, be naively integrated into restrictive languages. A comparison between constructive and restrictive specifications helps to uncover the intended semantics of the `any()` construct. We start with a tiny specification that is both given in B and in UML/OCL.

3.2 A Motivating Example

Figure 1 shows part of a Dispatcher-Depot scenario. A depot is a place to temporarily keep trains (e.g. during the night). For the purpose of our example, it is sufficient to know the number of trains which are currently at the depot (indicated by `no`). The task of a dispatcher is the management of depots, especially the dispatcher has to choose a depot where to place incoming trains (operation `addTrain()`). We assume a dispatcher to manage only two depots (`d1`,`d2`), furthermore we abstract from the fact that real world depots have a limited capacity.

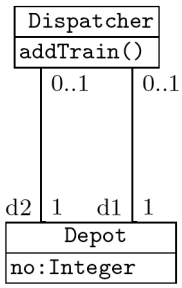
Figure 1 shows in its left-hand side a formalization of the Dispatcher-Depot example written in B whereas the right-hand side formalizes the same scenario using a UML/OCL specification.

The B specification starts with the description of train depots whose states are encoded by the state variable `no` of type `Integer`. The state of a dispatcher is given by the state variables `d1` and `d2` of type `Depot`. The specification of the operation `addTrain()` can be read as follows: It is always possible to invoke `addTrain()` (precondition is `true`) and upon termination of `addTrain()`, the number of trains in depot `d1` will be increased by 1 if `d1` had less trains than `d2` in the pre-state, otherwise the number of trains in depot `d2` is increased by 1.

The post-condition is constructive in the sense that it prescribes the behavior of `addTrain()` in an algorithmic way. Note, that the operator `:=` has to be read as assignment and thus the ordering of its arguments is crucial. In the

Machine Depot
Variables no
Invariant no ∈ ℤ

Machine Dispatcher(Depot)
Variables d1, d2
Invariant d1,d2 ∈ Depot
Operations
addTrain() ≙
PRE true
THEN
  IF no(d1) < no(d2)
  THEN no(d1) := no(d1) + 1
  ELSE no(d2) := no(d2) + 1
  END




```
context Dispatcher :: addTrain()
pre:  true
post: if d1.no@pre < d2.no@pre
  then d1.no = d1.no@pre + 1 and
    d2.no = d2.no@pre
  else
    d2.no = d2.no@pre + 1 and
    d1.no = d1.no@pre
  endif
```

Fig. 1. Constructive and restrictive specification in B and OCL

line `no(d1) := no(d1) + 1`, the value of the state variable `no` for `d1` (left-hand side) is updated with the value of this variable in the pre-state increased by one (right-hand side). The B specification also ensures that the number of trains is increased only for one of the two depots `d1`, `d2`; the number of trains in the other depot remains the same.

In the UML/OCL formalization, the declarations of the state variables are given in form of a UML class diagram. The lower part shows a restrictive specification of `addTrain()` written in OCL. The post-condition is structured the same way as the post-condition in the constructive B specification (`if-then-else`). Both specifications only differ in the `then`/`else` branches:

For example, the line `d1.no = d1.no@pre + 1` is not to be read as an assignment but just as a restriction that the state-variable `no` of `d1` has in the post-state the same value (`=`) as in the pre-state but increased by one. Note, that in contrast to the assignment operator used in the constructive B specification, the ordering of the arguments in the equality does not matter: the line `d1.no@pre + 1 = d1.no` would have expressed exactly the same.

There is another difference between constructive and restrictive specification that is illustrated in this tiny specification: The `then`-branch of the post-condition, for instance, covers the case where a train is added to the depot `d1` whereas depot `d2` remains untouched. If the latter fact is important (here it is, because an implementation of `addTrain()` would not be correct if it would, say, increase the number of both depots) it must be explicitly mentioned in the OCL specification (`d2.no = d2.no@pre`) whereas this is expressed in the B specifi-

cation automatically. For a deeper understanding of this problem (in literature known as the Frame problem) the interested reader is referred to [8].

3.3 Motivation for Non-determinism

The specification of `addTrain()` shown above is extremely detailed in the sense that for any given pre-state, the specification allows exactly one post-state. At a first glance, such specifications seem superficial because the implementation of the operation could have been given directly. This argument ignores the fact that the implementation and specification of a system usually reside on different levels of abstraction. An actual implementation for `addTrain()` would most likely use a much more detailed model of the system than would be derived by a refinement of the shown model. However, we use the term *implementation* in the rest of the paper as a synonym for the set of concrete pre-/post-state pairs that represent the behavior of the operation for the abstraction level given by the class diagram.

Normally, specifications are not as detailed as for `addTrain()` and intentionally leave more freedom to the implementations. Then, only a more liberal version of the specification would be appropriate, for example, that upon termination of `addTrain()` the number of trains of exactly one depot should be increased by one. This specification is less detailed because it does not prescribe which of the two depots will change its number of trains. Such a more liberal version can be easily formalized by a restrictive specification:

```
context Dispatcher :: addTrain ()
pre:  true
post: d1.no + d2.no = d1.no@pre + d2.no@pre + 1 and
      (d1.no = d1.no@pre or
       d2.no = d2.no@pre)
```

This OCL specification (basically) says that the sum of `no` for `d1` and `d2` is in the post-state increased by one compared to the pre-state.

How can this be expressed in a constructive specification using pseudo-code? If the specification language would only offer the constructs known from imperative programming languages, one had to decide which depot has to be taken (as in fig. 1). In order to cope with less detailed specifications, constructive specification languages offer constructs that allow a non-deterministic choice from a set of possible executions paths. The language B, for instance, offers `CHOICE-OR-END` as one construct to express non-determinism. The new specification for `addTrain()` could be expressed as follows:

Operations addTrain() $\triangleq$
PRE true
THEN
  CHOICE no(d1) := no(d1) + 1
  OR no(d2) := no(d2) + 1
  END

The meaning of the revised `addTrain()` specifications is best understood by evaluating them in a given pre-/post-state pair. As an example, the state pair $(S1, S2)$ where $S1 = (no(d1) = 2, no(d2) = 2)$ and $S2 = (no(d1) =$

$2, no(d2) = 3$) has been chosen.[4] Does this state transition conform to the two post-conditions?

Conformance to OCL specification The answer for the OCL specification is 'yes', because the state pair meets all restrictions made in the post-condition. Note, that the OCL specification would allow for the same pre-state also the post-state $S2 = (no(d1) = 3, no(d2) = 2)$.
If at least two post-states for the same pre-state are possible, then the behavior of a correct implementation cannot be predicted. In such cases, the OCL constraint is called an *under-specification* of the operation's behavior.

Conformance to B specification The answer for the B specification is also 'yes', because the construct `CHOICE` allows all implementations that realize the behavior given in one of the branches of `CHOICE`.
As for the OCL specification, the post-state $S2 = (no(d1) = 3, no(d2) = 2)$ would also be allowed. Both state transitions are possible due to the *non-determinism* of the construct `CHOICE`.

3.4 Mixing Restrictive and Constructive Specification Styles

Constructive specifications (illustrated above with a B specification, but another language such as ASM could have been used the same way) use pseudo-code to specify the behavior of operations in an algorithmic way. As seen in the first example, the behavior of an operation can easily be described by a constructive specification that leaves no room for variations among the implementations of the operation. If an equivalent specification should be given in a restrictive specification language such as OCL, then the Frame problem has to be addressed, which can result in a considerable explosion of the specification size.

On the other hand, constructive languages need constructs such as `CHOICE` to allow variations among possible implementations. In the case of the `CHOICE` construct, an implementation is seen to be correct if it correctly implements one of the branches.

Restrictive specification languages can easily express variations among the implementations by a weaker post-condition; this technique is called under-specification.

The construct `any()` offered by OCL can be seen as an attempt to combine the strengths of both specification paradigms. Thanks to the `any()` construct, an OCL specification can have the same structure as constructive specifications written in B, which can make them better understandable compared to equivalent, purely restrictive specifications.

The usage of `any()` in OCL is illustrated by a slightly extended version of the Depot-example. As shown in fig. 2, the trains at the depot are represented by a state-variable `ct` (in UML represented by an association between `Depot` and `Train`). The value of state variable `no` could be computed now as the cardinality of the set of trains denoted by `ct` and is, thus, omitted.

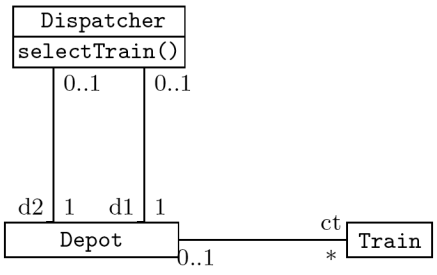
We consider a new operation `selectTrain()` on `Dispatcher` whose intended behavior is to select one train from one of both depots. It is assumed that

[4] Only the relevant part of the system state is given here.

Machine Train

Machine Depot(Train)
Variables ct
Invariant ct ⊆ Train

Machine Dispatcher(Depot)
Variables d1, d2
Invariant d1,d2 ∈ Depot
Operations
sel ← selectTrain() ≙
PRE ct(d1) ∪ ct(d2) ≠ ∅
THEN
  ANY t WHERE
    t ∈ ct(d1) ∪ ct(d2)
  THEN sel := t
  END




```
context Dispatcher ::
          selectTrain():Train
pre:  self.d1.ct->
      union(self.d2.ct)->
        notEmpty()
post: result =
      self.d1.ct->
        union(self.d2.ct)->any()
```

Fig. 2. Usage of any() in OCL

`selectTrain()` is only invoked in a state in which at least one depot has a train.

The B specification formalizes this informal specification in a natural way. The pre-condition encodes the availability of at least one train. In the post-condition, the return parameter of `selectTrain()` is declared by the variable `sel`. Moreover, an element t is selected non-deterministically from the set of available trains (this is done using the `ANY-WHERE` construct which is a generalized version of `CHOICE`) and then assigned to the return parameter `sel`.

The OCL specification has exactly the same structure. Instead of declaring a variable for the return parameter, OCL uses the predefined variable `result`. The post-condition states, that the value of `result` must be equal to `self.d1.ct->union(self.d2.ct)->any()`, which can be read as a non-deterministically chosen element from the set of trains available in depot $d1$ and $d2$.

Note, that the post-condition had to address the frame problem in order to become equivalent with the specification given in B. This could be done by extending the post-condition with `... and self.d1 = self.d1@pre and self.d2 = self.d2@pre ...` We have suppressed this part of the OCL post-condition here because it would distract us from the important part of the post-condition and our conclusions can already be drawn from the given version of the post-condition.

The semantics of both specifications is again best investigated with a concrete state transition. Let `selectTrain()` be invoked in a state where depot $d1$ has two trains $t1$,$t2$ and the depot $d2$ is empty. For the post-state, `selectTrain()` is assumed to return train $t1$.

This state-transition would clearly conform to the B specification. Analogously to `CHOICE`, the `ANY-WHERE` construct allow all implementations which

conform to one of the given choices (the state transition has taken the choice to assign train $t1$ to variable `t`).

The conformance to the OCL specification depends on the evaluation of the equation `result = self.d1.ct->union(self.d2.ct)->any()`, which can be simplified in the current situation to `t1 = {t1,t2}->any()`. According to the official semantics of `any()`, this can be evaluated to both *true* and *false* depending on the non-deterministic evaluation of `{t1,t2}->any()` to $t1$ or $t2$.

The example suggests the following *intended semantics* of `any()`: A state (or state-transition) conforms to a constraint *constr* containing `any()` if and only if among all alternatives for the evaluation of the `any()`-subterm there can be found at least one, such that the evaluation of the *constr* would result in *true*. Such a semantics would directly correspond to the semantics of the `ANY-WHERE` construct in B.

4 Improved Semantics for any() in OCL

Despite the clarification made in the last section on the role of `any()` in OCL specifications, the fundamental problems with the formal semantics of `any()` as described in section 2 are not fully solved yet. This section describes two approaches to overcome these problems.

4.1 Turning *eval* into an Evaluation Relation

The intended semantics of `any()` could be formalized by turning the evaluation function *eval* into an evaluation relation $eval_r$. On deterministic constructs, the relation $eval_r$ is exactly defined as the function *eval*. However, when the evaluation of a non-deterministic subterm allows multiple, non-deterministically chosen variants, the relation $eval_r$ results in all variants. This is possible because $eval_r$ is a relation and not a function like *eval* which has to decide for one of the variants. A state conforms to a constraint if and only if its evaluation in that state by $eval_r$ yields at least for one variant the result *true*.

Although $eval_r$ formalizes the intended semantics of non-deterministic constructs, it has some deficiencies that prevent its adoption in practice.

Firstly, the evaluation of a constraint in a given state can become exponentially complex if non-deterministic terms are nested. Note, that the evaluator had to handle all possibilities for an evaluation instead of just one result in case of deterministic evaluation.

Secondly and more important, $eval_r$ breaks with the traditional way in logic to define the semantics of specification languages. As illustrated in section 2 with the `foo2()` example, by adopting the $eval_r$ semantics for OCL, we would sacrifice common basic logical laws, for instance that `=` is a reflective relation so that expressions of form X `=` X can be simplified to `true`. Consequently, we would lose the tool support gained for OCL due to the fact that OCL is based on first-order logic.

4.2 Transformational Approach

The second proposal to define a semantics for `any()` is in terms of a transformation from non-deterministic specifications to deterministic ones for which the

official OCL semantics can be applied. Thus, the drawbacks of the $eval_r$ proposal do not apply here.

However, the transformational approach has some other drawbacks. The resulting formula is more complex than the original one.[5] A second drawback is that the transformation is not always applicable. Fortunately, this seems to be not a serious restriction in practice and the transformations can handle, for instance, all occurrences of `any()` in the UML metamodel.

The Algorithm As pointed out in the $eval_r$ approach, the intended meaning of non-deterministic constructs is to take all possible evaluations into account.

Let *constr* be a constraint that contains a term $t \equiv$ *set*`->any()` at a position *pos* (indicated by *constr*`(`t^{pos}`)`). Following the intended semantics of `any()` we know that *constr*`(`*set*`->any()`^{pos}`)` is evaluated in a given state to *true* if and only if there exists in the evaluation of *set* an element *o* such that *constr*`(`o^{pos}`)` is evaluated to *true* or, in the case that *set* is evaluated to the empty set, that *constr*`(undef`^{pos}`)` is evaluated to *true*. This justifies transformation of the constraint as shown in fig. 3.

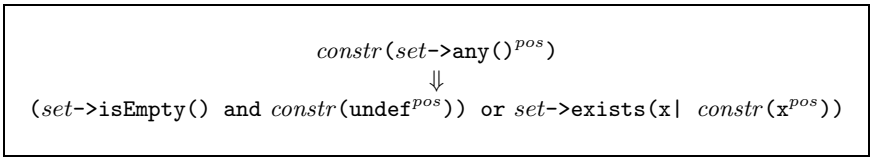


Fig. 3. Substitution of any() by deterministic constructs

Informally speaking, it is first tested whether *set* evaluates to the empty set and in this case the `any()`-term *set*`->any()` occurring in *constr* is substituted by `undef` or, otherwise, the `any()`-term in *constr* is substituted by a variable `x` which is introduced outside *constr* by an exists quantifier over *set*. Note, that the subterm *set* is moved from inside to outside of *constr*. This is only possible if *set* does not contain any variables introduced by iteration operations such as `forAll`, `exists`, `select`, etc., because the transformation would then result in a syntactically incorrect OCL term. For example, if the transformation were to be applied mechanically on the constraint

```
Set{1,2}->forAll(y| Set{y}->any() > 1)
```

then it would yield

```
(Set{y}->isEmpty() and Set{1,2}->forAll(y| undef > 1)) or
Set{y}->exists(x| Set{1,2}->forAll(y| x > 1))
```

what is a syntactically incorrect OCL term because the variable `y` in `Set{y}` is not declared.

[5] This is, on the other hand, also an argument for the simplicity and readability made possible by `any()`.

Despite the restricted applicability, the transformation defined in fig. 3 can successfully be applied on all examples discussed in this paper.

Example foo1():

```
context Foo :: foo1():Integer
post: result = Set{1,2}->any()
                ⇓
context Foo :: foo1():Integer
post: (Set{1,2}->isEmpty() and result = undef) or
       Set{1,2}->exists(x| result = x)
```

Since the set denoted by `Set{1,2}` is not empty, the result of the transformation could be simplified to

```
context Foo :: foo1():Integer
post: Set{1,2}->exists(x| result = x)
```

and even further simplified to

```
context Foo :: foo1():Integer
post: Set{1,2}->includes(result)
```

Example foo2():

The post-condition for `foo2()` contains two `any()`-terms and requires applying the transformation twice. For brevity, the result of the transformation has already been simplified (`isEmpty()` branches have been removed).

```
context Foo :: foo2():Integer
pre:  true
post: if (Set{1,2}->any() = Set{1,2}->any())
       then result = 1
       else  result = 2
       endif
             ⇓
context Foo :: foo():Integer
pre:  true
post: Set{1,2}->exists(x1| Set{1,2}->exists(x2|
       if (x1 = x2)
       then result = 1
       else  result = 2
       endif))
```

Note that this specification allows the implementation to return both 1 and 2. The first case is made possible by assigning `x1` to 1 and `x2` to 1, the latter case by assigning `x1` to 1 and `x2` to 2.

Example selectTrain():

```
context Dispatcher :: selectTrain():Train
pre:  self.d1.ct->union(self.d2.ct)->notEmpty()
post: result = self.d1.ct->union(self.d2.ct)->any()
                ⇓
context Dispatcher :: selectTrain():Train
pre:  self.d1.ct->union(self.d2.ct)->notEmpty()
post: self.d1.ct->union(self.d2.ct)->exists(x| result = x)
```

This can be simplified to

```
context Dispatcher :: selectTrain():Train
pre:  self.d1.ct->union(self.d2.ct)->notEmpty()
post: self.d1.ct->union(self.d2.ct)->includes(result)
```

5 Conclusion

Currently, the semantics of non-deterministic constructs in OCL is not clearly defined. The semantic foundation of non-deterministic constructs given in the official language description can be easily misunderstood, which leaves room for different interpretations. None of the current OCL tools is able to handle non-deterministic constructs properly, which is a sign for the poor understanding of such constructs. In practice, use of non-deterministic constructs is avoided, or they are only used in cases in which deterministic evaluation is ensured, such as the transformation of a singleton set to an object.

We have pointed out that non-deterministic constructs are very useful and even necessary in constructive specification languages such as B or ASM. The language OCL tried to adopt these constructs without paying attention to the characterization of OCL as a restrictive specification language. The comparison of OCL with constructive languages has revealed the intended semantics of non-deterministic constructs. As illustrated by examples, specifications in constructive languages using non-deterministic constructs can easily be rewritten in OCL without using non-deterministic constructs. In order to describe non-deterministic behavior, restrictive specification languages such as OCL offer the technique of under-specification.

Nevertheless, the non-deterministic construct `any()` allows the user to write OCL specification in a more 'constructive style'. This can make specifications more accessible for users with a strong background in programming. Since we were able to formally define the semantics of `any()` in terms of a code transformation, the construct `any()` could be easily integrated into other restrictive languages such as Z, JML, Eiffel. Such an integration could make these languages more usable and, thus, increase the acceptance of formal methods, especially for people who are used to describing the behavior of systems in a constructive way. Seen this way, OCL's often misunderstood `any()` construct has brought some innovation into the realm of restrictive specification languages.

References

1. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.
2. UML 2.0 Infrastructure Specification – OMG Adopted Specification. OMG Document ptc/03-09-15, Sep 2003.
3. UML 2.0 Superstructure Specification – OMG Adopted Specification. OMG Document ptc/03-08-02, Aug 2003.
4. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
5. Overview on current OCL tools. www.klasse.nl/ocl/ocl-services.html.
6. Dan Chiorean, Maria Bortes, Dyan Corutiu, and Radu Sparleanu. UML/OCL tools - objectives, requirements, state of the art – The OCLE experience. In *Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques NWPER'2004*, number 34 in TUCS General Publication, Turku, pages 163–180, 2004.
7. Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.
8. C. Morgan. *Programming form Specifications*. Prentice Hall, 1994.