Panel Discussion: Proposals for Improving OCL

Achim D. Brucker[1], Tony Clark[2], Carolina Dania[3], Geri Georg[4], Martin Gogolla[5], Frédéric Jouault[6], Ernest Teniente[7], and Burkhart Wolff[8]

[1] SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com
[2] Department of Computer Science, Middlesex University, London, UK
t.n.clark@mdx.ac.uk
[3] IMDEA Software Institute, Madrid, Spain
carolina.dania@imdea.org
[4] Computer Science Department, Colorado State University, Fort Collins, USA
georg@cs.colostate.edu
[5] Database Systems Group, University of Bremen, Germany
gogolla@informatik.uni-bremen.de
[6] LUNAM, L'Université Nantes Angers Le Mans
TRAME team, ESEO, Angers, France
frederic.jouault@eseo.fr
[7] Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – BarcelonaTech
teniente@essi.upc.edu
[8] Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France
CNRS, 91405 Orsay, France
burkhart.wolff@lri.fr

Abstract. During the panel session at the OCL workshop, the OCL community discussed, stimulated by short presentations by OCL experts, potential future extensions and improvements of the OCL. As such, this panel discussion continued the discussion that started at the OCL meeting in Aachen in 2013 and on which we reported in the proceedings of the last year's OCL workshop.
This collaborative paper, to which each OCL expert contributed one section, summarises the panel discussion as well as describes the suggestions for further improvements in more detail.

1 Introduction

While OCL is nearly 20 years old [6], it is still an evolving language and there is an ongoing effort in academia to improve it. This is also witnessed by the constant updates to the official OMG standards and the current standardisation efforts that will eventually results in OCL version 2.5. Already as a follow up of the last OCL workshop, a number of OCL experts met in November 2013 in Aachen to discuss possible improvements of the OCL (see the report on the OCL meeting in Aachen in the proceedings of the OCL workshop 2013 [2]).

The panel session provided a platform for the OCL community to discuss the presented proposal for improving the OCL as well as to discuss the general

future of textual modelling. The following sections, each of them contributed by one expert of the field, discuss the different areas for improvements that were discussed during the panel session.

2 Frame Conditions for OCL

Achim D. Brucker. Traditionally, OCL operation contracts do only specify the intended changes to the system state. In general, there is no guarantee that other parts of the system remain unchanged. In particular, the default post condition `true` allows arbitrary changes to the system state.

We suggest to introduce a new method, called `_->modifiesOnly()`, that allows to explicit specify frame conditions, i. e., what can be modified by an OCL operation.

2.1 Motivating Example

When using contracts, or pairs of preconditions and postconditions for state transition there arises the need to specify *exactly* which parts of the system are allowed to be modified and which have to stay unchanged, i. e., we have to specify the frame property of the system. Otherwise, arbitrary relations from pre-states to post-states are allowed. For most applications this is too general: there must be a way to express that parts of the state *do not change* during a system transition, i. e., to specify the frame properties of system transition. As

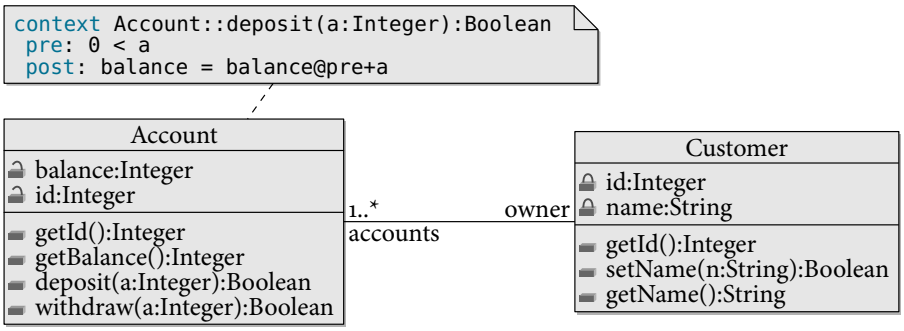

Fig. 1. Consider a state transition constrained by the operation specification for the operation `deposit`. Obviously, only the attribute `balance` of one specific object should be changed, but how can this be specified?

an example, consider Figure 1 with an particular focus on the specification of the operation `deposit` of the class `Account`. This specification only describes which part of the system should change, i. e., the balance of the context object (which is an `Account` object) should be increased. But this is not specified, which parts of the system should remain unchanged, e. g., the `id` of the context object.

One solution to solve this frame problem would be an implicit invariability assumption on the meta-level which would somehow express "all things that are not changed explicitly remain unchanged." But this is neither formal nor precise and thus not usable within a formal framework for object-oriented specifications.

Another possibility is to enumerate, in the postcondition of the operation, all path expressions that should remain unchanged, e. g., in our example a first attempt to do so would be:

```
context Account::deposit(a:Integer):Boolean
  post: balance = balance@pre+a
  post: id = id@pre
  post: owner = owner@pre
  post: owner.id = owner@pre.id@pre
  post: owner.name = owner@pre.name@pre
```

But this is also not sufficient, as it would still not describe if objects not related to our context object (of type `Account`) must remain unchanged or not. Enumerating all classes (and attributes) using static path expressions (e. g., `Customer::name = Customer::name@pre`) is tedious and moreover leads to contradictions if the `name` attribute of the owner of the context object should be changed.

Our Proposal. This framing problem is well-known (one of the suggested solutions is,e. g., [5]). We suggest to introduce an OCL method that explicitly allows to specify what might be changed during a system transition. We define

```
(S:Set(OclAny))->modifiesOnly():Boolean
```

where `S` is a set of objects (i. e., a set of `OclAny` objects). This also allows recursive operations collect the set of objects that are potentially changed by a recursive function. Obviously, similar to `@pre` the use of `->modifiesOnly()` is restricted to postconditions.

In our formalisation, called Featherweight OCL [3], we encode the set `S` as a set of object ids (oid). The semantics of the `_->modifiesOnly()` operator is defined such that for any object whose oid is *not* represented in `S` and that is defined in pre and post state, the corresponding object representation will not change in the state transition. A simplified presentation is as follows:

$$I[\![X\texttt{->modifiesOnly()}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } X' = \bot \vee \text{null} \in X' \\ \lfloor \forall\, i \in M.\ \sigma\ i = \sigma'\ i \rfloor & \text{otherwise}. \end{cases}$$

where $X' = I[\![X]\!](\sigma,\sigma')$ and $M = (\text{dom}\ \sigma \cap \text{dom}\ \sigma') - \{\text{OidOf}\ x |\ x \in \lceil X' \rceil\}$. Thus, if we require in a postcondition `Set{}->modifiesOnly()` and exclude via `_.oclIsNew()` and `_.oclIsDeleted()` the existence of new or deleted objects, the operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \vDash X\texttt{->excluding}(s.a)\texttt{->modifiesOnly()}$ and $\tau \vDash X\texttt{->forAll}(x|\texttt{not}(x \doteq s.a))$, we can infer that $\tau \vDash s.a \triangleq s.a\,\texttt{@pre}$.

3 Extending OCL with Functions

Tony Clark. The current OCL standard does not support *functional abstraction*. This is surprising given the origins of OCL and its relationship, with respect to extensive support for collection processing, to functional programming languages such as ML and Haskell. OCL can currently be used to specify operations on classes. It would be useful to extend this notion so that OCL could be used to define a self supporting, albeit high-level and side-effect free, executable system that could be used for a variety of purposes including scripting, simulation, model management, *etc.* Adding functions to OCL is a step in this direction.

In this regard, a *functional abstraction* in OCL will provide a new type of expression that defines an anonymous function comprised of a sequence of named arguments and a body, which is any OCL expression. The denotation of such an expression is a *function* that can be applied to the requisite number of argument values causing the function definition body to be evaluated. Since a function is a value it can be named in the usual way, for example by passing it as an argument to another function or (equivalently) binding in a **let**-expression. Free variables within the body of a function definition will exhibit *lexical scoping*, meaning that the life-time of the function, and any associated free variable values, may outlive that of the binding scope in which it is defined. Since the name of a function is not an intrinsic part of its definition, recursive functions are to be established using a new binding mechanism provided by `let` that is designated as *recursive*.

Having outlined above the characteristics of functions, their use within OCL is motivated as follows:

abstraction Currently OCL lacks a mechanism for abstracting patterns of definitions and then reusing them throughout a system specification. This may take the form of a collection of domain specific functions that provide, for example, arithmetic calculations. Furthermore, the higher-order aspect of functions will facilitate patterns over functions, for example by defining calculations involving sorts where the sort-relationship (alpha-sort, numeric-sort, ascending, descending, *etc.*) is passed as an argument.

modularity Current OCL specifications can be long-winded where expressions contain a great deal of detail. Functions, especially locally defined functions, can help to reduce the complexity both in terms of size and readability. Functions allow parts of a specification can establish a collection of private reusable abstractions. Functions can be the basis of defining both general-purpose and domain-specific library modules for OCL.

iteration OCL provides a string support for processing collections. The iteration processing expressions are built-in to the OCL language when this is not necessary. They can all be defined in terms of a small number of primitive collection operations and recursive functions (as demonstrated by functional languages whose libraries contain a much larger range of collection operators). Languages should strive for both semantic universality and semantic parsimony with regard to their intended domain; currently OCL provides neither.

3.1 Proposal

This section proposes the addition of anonymous function definitions and functions, and associated language support, to OCL. This section provides a brief overview of how this might be achieved.

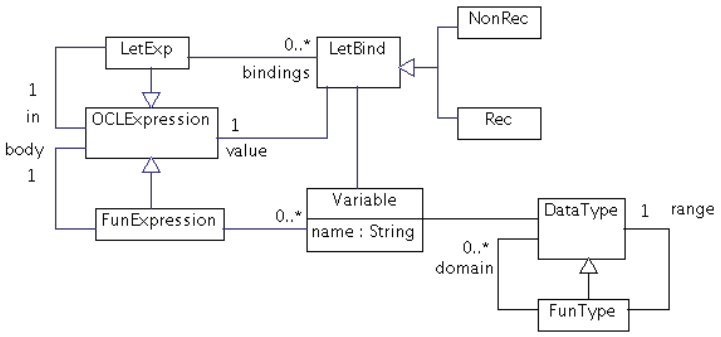

Fig. 2. Abstract Syntax Extension

Types Figure 2 shows the proposed extensions to the OCL abstract syntax model that are necessary to support function definitions. A new type `FunExpression` is introduced as a sub-class of `OCLExpression`, it has any number of (ordered) arguments defined as variables, and a body. The existing `LetExp` class is extended to allow two different types of binding: recursive and non-recursive. The `DataType` class is extended to produce a new data type called `FunType` whose domain and range types describe the argument and body types of a function respectively.

Concrete Syntax OCL functions are supported by a small concrete syntax extension. A function expression including domain and range types can be written as follows (assuming the availability of a function `sqrt`):

```
fun(x:Integer,y:Integer):Integer
  sqrt(x*x + y*y)
end
```

We can name the function:

```
let
  distance = fun(x:Integer,y:Integer):Integer sqrt(x*x + y*y)
  end
in distance(100,200)
end
```

which suggests the following sugar:

```
let distance(x:Integer,y:Integer):Integer = sqrt(x*x + y*y)
in distance(100,200)
end
```

At the top-level of a specification we might allow `distance` to be available everywhere:

```
let distance(x:Integer,y:Integer):Integer = sqrt(x*x + y*y);
```

In the above definition, the name `distance` will not refer to the function being defined (if anything it will refer to a definition in a surrounding scope). To achieve a recursive function, an extra keyword is used:

```
let rec
  size(s:Sequence(T)):Integer = if s->isEmpty
                                then 0
                                else 1 + size(s->rest());
```

3.2 Examples

To see how higher-order features of functional-OCL can be used to good effect, consider the case of OCL without built-in iteration. The `select` expression can be achieved using a function called `select` that is defined in the context of a polymorphic type `Sequence(T)`. The function `select` takes a function `q` as an argument; `q` acts as a predicate on each element of the collection. The function `select` recursively processes the collection and returns a collection containing only those elements that satisfy `q`:

```
context Sequence(T)::select(q:(T)->Boolean):Sequence(T) =
  let s:Sequence(T) = self->rest()->select(q)
      x:T = self->first()
  in if q(x)
     then s->prepend(x)
     else s
     end
  end
```

Now any occurrence of `S->select(e | p)` can be translated to: `S.select(fun(e) p end)` and all other OCL iteration constructs can be treated the same way. This significantly reduces the number of semantic primitives for OCL and provides a basis for new collection processing operations based on higher-order functions, for example:

```
context Sequence(T)::foldr(g:(T,T')->T',x:T'):T' =
  if self->isEmpty
  then x
  else g(self->first(),self->rest().foldr(g,x))
  end
```

Once defined, this can be used as the basis of many different sequence operations:

```
context Sequence(Boolean)::allTrue():Boolean =
  self.foldr(and,true)
context Sequence(Boolean)::anyTrue():Boolean =
  self.foldr(or,false)
context Sequence(Integer)::sum():Integer     =
  self.foldr(+,0)
context Sequence(Integer)::product():Integer =
  self.foldr(*,1)
context Sequence(T)::size():Integer          =
  self.foldr(fun(x) x + 1 end,0)

context Sequence(Sequence(T))::concat():Sequence(T) =
  self.foldr(fun(l1,l2) l1->append(l2) end,Seq{})
context Sequence(T)::reverse:Sequence(T) =
  self.foldr(fun(x,l) l->append(Seq{x}) end,Seq{})
```

4 Implicit Strict Downcasts in OCL Collection Operations

Martin Gogolla. Current OCL allows to select elements of a particular type from a heterogeneous collection and to apply subtype specific operations to the selected elements.

```
Set{4,'VII','IV',7}->
  selectByKind(Integer)->
    collect(i | i*i)
==> Bag{16,49} : Bag(Integer)
```

As an example, consider the above evaluation on a heterogeneous collection with `Integer` and `String` elements. Due to the relatively new operation `selectByKind` this task can be formulated in a more condensed way than in older OCL versions.

```
Set{4,'VII','IV',7}->
  select(x:OclAny | x.oclIsTypeOf(Integer))->
    collect(x | let i:Integer=x.oclAsType(Integer) in i*i)
==> Bag{16,49} : Bag(Integer)
```

As shown above, evaluations of this kind were possible in OCL from the very beginning by employing `select`, type assertions, `collect`, and type downcasts, however more notational overhead was needed when compared to the formulation with `selectByKind`.

The proposal that we put forward here is to reduce the notational overhead even more by allowing explicit downcasts from more general types to more special types in collection operations by explicitly giving a subtype to a variable that is used in the collection operation.

```
Set{4,'VII','IV',7}->
  collect(i:Integer | i*i)
==> Bag{16,49} : Bag(Integer)
```

Starting from types `S` and `G` with `S<G` and a term `COL` evaluating to a collection of type `Collection(G)`, the general translation schema for such explicit downcasts in collection operations would look as indicated below: the central idea is that a call for `colOp` is replaced by a `select` call and a `colOp` call; the variable `s` is typed through the more special type `S` and the OCL expression `expr[s]` uses `s` in contexts where the more special type `S` and not the more general type `G` is expected; the operation `colOp` can be any collection operation, not only as in the above example the collection operation `collect`.

```
COL->colOp(s:S | expr[s])
==>
COL->
  select(x | x.oclIsTypeOf(S))->
    colOp(g:G | let s:S=g.oclAsType(S) in expr[s])
```

In particular applications of this construct in the context of type generalization seem to be useful. For example, assume we have `Female<Person, Male<Person`, the following evaluation would be possible.

```
Set{ada,bob,cyd,dan,eve}->
  collect(f:Female | f.husband.firstName)
==> Bag{'Dan','Dan'}
```

The discussion at the workshop brought up reservations about the use of the colon `:` at the crucial point of giving a subtype to the variable. In order to avoid accidental use of the subtyping mechanism, using a new, differentiating syntax was discussed. Some proposals are indicated below. The lasts syntax proposal employing brackets are referring to the syntax proposed for patterns matching (inspired from Haskell).

```
Set{ada,bob,cyd,dan,eve}->
  collect(f<:Female | f.husband.firstName)

Set{ada,bob,cyd,dan,eve}->
  collect(f@Female{} | f.husband.firstName)

Set{ada,bob,cyd,dan,eve}->
  collect(f:Female{} | f.husband.firstName)
```

5 Active Operations for OCL

Frédéric Jouault. Before its 2.0 version (when it was still defined as a part of UML [8]), OCL only had three kinds of constraints: `inv` for classifier invariants,

as well as `pre` and `post` respectively for preconditions and postconditions of operations[9]. Starting with version 2.0, additional kinds of constraints appeared: `body` to implement (side-effect free) operations, as well as `init` and `derive` respectively to specify initial values, and to implement derived features.

A derived feature is a feature (attribute or association end), which has a value that is computed from the values of other features. The fact that a feature is derived is specified in a class diagram (e.g., in UML), not in OCL. In general, the value of a derived feature is specified in OCL as an invariant. However, the computation of the value of a derived feature is not always trivial from an invariant. Consequently, `derive` constraints were introduced in the OCL specification as a special case of invariants. They work by specifying an expression that evaluates to the value of the derived feature. The computation of the value of a derived feature becomes as simple as evaluating that expression.

Although the more recent `derive` kind of constraint is useful, it notably does not address the 3 following issues that arise when working with derived features:

- **Changeability** is limited to read-only access. Writable derived features must typically be achieved by actual implementation (e.g., in Java), not by modeling in OCL.
- **Observability** is generally not possible because derived features are computed by evaluating the specified OCL expressions, which typically happens on-demand. Arguably, this is more an implementation issue than a modeling one. However, it is a real problem for modeling tools. For instance, a model editor that displays the value of a derived feature cannot directly listen for its changes.
- **Direct use of invariants** to specify derived features should also be possible, but is generally not supported by tools. When `derive` is used instead of `inv`, the person writing the constraint must decide how to compute the derived features. Ideally, tools should be able to do this from more general invariants.

In this section, we propose to extend OCL with active operations [1]. Basically, active operations enable incremental synchronization of collections. In some cases, bidirectionality is even possible.

Here are three concrete benefits of integrating active operations in OCL:

- **Changeability** is achieved by relying on bidirectionality of active operations. This works independently of whether `inv` or `derive` is used.
- **Observability** is possible because active operations incrementally update derived features as soon as there is a change in the values they depend on. This is also independant of the use of `inv` or `derive`.
- **Direct use of invariants** becomes possible without having to rewrite them into `derive` constraints (whether manually or automatically). This means that `inv` may be used instead of `derive`.

Active operations perform their work by producing side-effects on models: some collections get updated when other collections are changed. We believe they

[9] `def` "definition" constraints enable expression reuse but do not constrain models.

are nonetheless compatible with OCL because: only "when OCL expressions are evaluated, they do not have side effects" [9], but active operations do not work by evaluating OCL expressions. With active operations, OCL constraints are used in way that is similar to how constraints are used in constraints programming: they specify the form of desired solutions, not how to compute them.

5.1 Motivating Example

Consider Figure 3: a *Transporter* handles several *Transport*s, each associated to a *Truck* and a *Driver*, which are two kinds of *Resource*s. The set of all resources used by a *Transporter* is captured as the *resources* association. Finally, a *Transporter* has three derived features: *derivedResources*, *trucks*, and *drivers*. Note that from a modeling point of view *derivedResources* is redundant with *resources*. However, from a pedagogical point of view we need both derived and non-derived versions of the same relation.

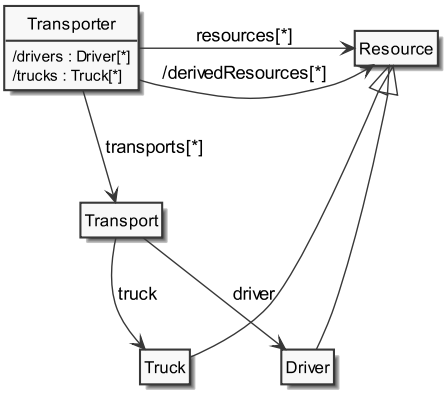

Fig. 3. Transporter class diagram

Listing 1.1 gives a set of OCL constraints over the class diagram depicted in Figure 3. Constraint C1 specifies with an invariant that all *derivedResources* of a transporter must be used in at least one transport. Constraint C2 specifies how the *derivedResources* derived feature can be computed. One can observe that C1 logically implies C2. Nonetheless, C2 must generally be specified so that tools may actually be able to compute the value of the derived feature. With active operations, the more general C1 is enough and C2 is redundant. Nonetheless, C2 would also work without C1. Note that in this case *derivedResources* cannot trivially be made writable because adding a resource to a transporter may then require the creation of a new instance of class *Transport*. This may be captured in the class diagram by specifying *derivedResources* as read-only. It should be noted that UML also provides a specific built-in mechanism for derived unions,

which is roughly equivalent to what we are doing here with C1. However, UML specifies that derived unions must be read-only whereas we can do better with active operations.

Similarly, constraint C3 specifies with an invariant that non-derived *resources* correspond to the union of derived *drivers* and *trucks*. Constraint C4 and C5 respectively specify expressions that can be used to directly compute the values of *drivers* and *trucks*. One can observe that C3 logically implies C4, and C5. Nonetheless, C4 and C5 must generally be specified so that tools may actually be able to compute the values of the derived features. With active operations, the more general C3 is enough and C4 and C5 are redundant (but would still work). Additionally, C3 is usable in both directions: *resources*, *drivers*, and *trucks* are all writable features. Moreover, even if C3 did not exist, C4 and C5 would be usable bidirectionaly to update *resources* when either *drivers* or *trucks* was updated.

```
-- [C1] all derivedResources must be used by transports
context Transporter inv:
self.derivedResources = self.transports.driver->union(
        self.transports.truck)

-- [C2] implementation of /derivedResources derived feature
context Transporter::derivedResources : OrderedSet(Resource)
derive: self.transports.driver->union(self.transports.truck)

-- [C3] resources are the union of drivers and trucks
context Transporter inv:
self.resources = self.drivers->union(self.trucks)

-- [C4] implementation of /drivers derived feature
context Transporter::drivers : OrderedSet(Driver)
derive: self.resources->select(e | e.oclIsKindOf(Truck))

-- [C5] implementation of /trucks derived feature
context Transporter::trucks : OrderedSet(Truck)
derive: self.resources->select(e | e.oclIsKindOf(Truck))
```

Listing 1.1. OCL constraints for the class diagram of Figure 3

We have seen above that with active operations, not only are C1 and C3 enough, but they are also enough for the tools to be actually able to compute the values of derived features. Concretely, here is how active operations work for C3:

- **Initialization** consists in setting *drivers* and *trucks* to appropriate values by doing something similar to what C4 and C5 specify, but by only relying on information given in C3. Additionally, the active operations engine starts to listen for changes in either *resources*, *drivers*, or *trucks*.
- **Synchronization** is triggered whenever a change is performed, and the model is considered to be in a stable state only after it is done. If *resources*

is modified, then either *drivers* or *trucks* is updated with the new element. If either *drivers* or *trucks* is modified, then *resources* is updated with the new element. Should these features be ordered, active operations may preserve ordering (e.g., by considering that union is equivalent to concatenation).

Once synchronization has ended, the model is again in a stable state, and the invariant satisfied.

5.2 Conclusion

Active operations address concrete shortcomings of OCL when used with derived features. They are compatible with its semantics, even though they require a different execution engine. Algorithms proposed in [1] can be used to implement active operations, and cover most OCL operations on collections. We do not foresee any issue with missing ones.

6 Purpose-specific Fragments of OCL

Ernest Teniente.[10] An important direction on the improvement of OCL should be devoted to analyze how can we make this language broadly used in industry. In this proposal, we suggest to identify purpose-specific fragments of OCL, each of them devoted to a different goal in software development, as a significant step in this direction. Having purpose-specific fragments of OCL would help its learning and understanding while showing up the benefits of its use.

6.1 Motivation

OCL is aimed at defining all relevant aspects of a specification that cannot be stated diagrammatically. It is a formal language, intended to be easy to read and write and it can be used for a number of different purposes: as a query language, to specify invariants in UML class diagrams, to describe pre- and post condictions on operations and methods, to describe guards, to specify target sets for messages and actions, etc. [7].

OCL is proposed to be used in software development, mainly at the initial stages of this process, because it is intended to fill the gap between natural and classical formal languages being understandable but formal at the same way; and it is expected to be widely used in industry because of the advantages it provides to the automation of code generation or to automated reasoning. This is particularly important in the context of model-driven development.

Providing an answer to the question *"How can OCL be improved?"* should take all these issues into account. So, the aim of this contribution is not to propose some additional, missing, feature of the language to make it better in

[10] This work has been partially supported by the Ministerio de Ciencia e Innovación under project TIN2011-24747

some sense but trying to contribute to the debate of why OCL is not used (almost) in industry and to propose a possible solution to it.

There are two possible reasons for that: either the industry does not build (UML) models or OCL is too complex already to be understood by people from industry. In the first case, there would still be a long way to go as far as modeling in industry is concerned and making the language more expressive would not solve the problem. In the second, it seems clear that considering additional features of the language would not help people to better understand OCL.

Some issues allow to illustrate the difficulties to understand OCL. For instance, when learning about the use of OCL expressions in UML models in the OCL specification document [7], we find the following sentence: "everywhere in the UML specification where the term expression is used, an OCL expression can be used (...), but other placements are possible too. The meaning of the value, which results from the evaluation of the OCL expression, depends on its placement within the UML model". Does it mean that we need to read the whole UML specification also to know when an OCL expression can be used? How do we know the different values that an expression can take depending on its placement? How can we easily learn that?

Additionally, some aspects are difficult to explain or may even look like contradictory. A couple of examples follow. Being OCL a declarative language, is the *iterate* construct declarative? How can we justify that? When should *tuple types* be used? What are they useful for? Which is the relationship between the expressive power of OCL and that of well-known languages such as relational algebra or first-order logic?

Under these considerations, and bearing in mind that our purpose is getting OCL to be used in industry, my guess is that we should not actually extend OCL further but trying to simplify or to structure it in such a way that it can be better understood. This is further discussed in the next section.

6.2 Purpose-specific Fragments of OCL

One way to improve the understanding of OCL, thus facilitating its adoption by industry, would be to identify *purpose-specific* fragments of the language. Each fragment should be devoted to a particular purpose or use of the language in software development. So, we could identify a fragment to define invariants in UML class diagrams, another one to specify pre- and postconditions of operation contracts, a third one to state model transformations, etc.

Different fragments would share several OCL operators but probably none of them would require the whole set of actual operators because of its particular purpose. Note that we are not advocating for simplyfing the language nor for removing some operators. We are just proposing to structure the language in such a way that understanding each fragment is easier than understanding the whole OCL. Moreover, knowing that the fragment is aimed at an specific purpose would also facilitate knowing the formal semantics of the language and the comparison with other languages for the same purpose.

The definition of the expressions and the operators of each fragment should be performed in an incremental way, going from the basic concepts and the most common patterns found in its purpose to the most complex and specific issues. Thus, for instance, in the fragment devoted to specify invariants in UML class diagrams one could start by showing how to build expressions that specify invariants tied to a contextual instance and obtained through the navigation of binary associations and, afterwards, proceed with more difficult cases like explaining the particular usage of the *allInstances* operator (i.e. when is it strictly required and why) or the navigation through n-ary associations.

Having a purpose-oriented structure of the language would allow also identifying the tools that allow the automation of software development with that purpose in mind. This is particularly important because without the existence of such tools it will be very difficult for industry to adopt the OCL language. In fact, the best way to convince industry about the usefulness of the use of OCL would be to show the economical benefits they would get and also the improvement on the quality of the final product obtained, and this can only be achieved by means of practical tools.

Finally, and is it clearly happens in the Java community, we would also need books so that people can do self-learning of each specific fragment of OCL. Right now, there are only a very few books, rather introductory, explaining how the elements of the OCL language can be used to complement UML models [4, 10, 11]. Moreover, they are ten years old already and aimed at covering the most common usages of OCL thus being too general as far as self-learning for an specific purpose is concerned.

6.3 Conclusion

Simple is better, definitely. So, if we want the OCL language to be widely used in industry we need to structure it in such a way that it is easily understandable, easy to learn and so that the benefits it provides to software development are out of discussion. One way to achieve this is by considering purpose-specific fragments of the language as advocated in this paper. Tools and books for each of the fragments are also required.

7 Patterns in OCL

Burkhart Wolff. Pattern-Matching is a widely used and well-known concept in functional programming leading to concise and readable code. They are particularly valuable for defining model-transformers and compilers, a domain where OCL is prominently used.

With the advent of Tuples (called *records* in the functional programming literature) we could also introduce pattern-matching wherever variables were bound, so in definitions of recursive functions, quantifiers, select-operators, ...

Moreover, we suggest the concept of *shadow classes* which can be associate to each class-definition `A` (allowing objects in a state) a shadow - tuple `A{a, ... , z}` (so: a value) that is amenable to pattern matching.

7.1 Pattern Matching in Collection Types

OCL possesses hidden second-order combinators, implicitly accepting a lambda buried under first-order notation. These are:

```
->iterate        ->exists      ->forall        ->select
->collect        ->any         ->isUnique.
```

For all these constructs, we propose to allow:

```
S->select( PATTERN | P (x1,...,xn))
```

or in the general case:

```
S->select( PATTERN1 | P1 (x1,...,xn)
         @ ...
         @ PATTERNm | Pm (x1,...,xmn))
```

For example:

```
S->select(Seq{_, 3, a, b, ...} | a >= 15 and a = b)
```

which filters from a Collection of Sequences integers those who have a 3 as second argument, and where the third argument is larger 15 and identical with the forth argument. It can be seen as shortcut for:

```
S->select(X | X->nth(2) = 3 and X->nth(3) >= 15
                and X->nth(3) = X->nth(4))
```

Similarly, a construction like:

```
S->select(Set{3, a, b, ...} | a >= 15 and a = b)
```

could be seen as shortcut for:

```
S->select(X | X->includes(3) and
              X->exists(a b | a >= 15 and a = b))
```

With respect to Tuples (called usually *records* in functional programming languages), the following notations are possible:

```
S->select(Tuple{name='mueller',sex=male,age=x, ...} |
                                                  x >= 21)
```

Note that to be on the safe side, we propose to allow the ... notation for unused labels in a tuple, but allow the pattern-match notation only when the tuple type can be completely inferred.

7.2 Shadow Tuples of Classes

The power of the pattern-matching mechanism is further increased if a seamless transition between objects and corresponding tuples is supported. For example:

```
class Employee is Person
+ salary : Integer[0..1]
+ dept_id : Integer [1]
end
```

induces the implicit declaration of the *shadow-tuple*:

```
Employee{salary : Integer;  dept_id : Integer;
         sex: Sex; name: String }
```

(where `Person` provides the remaining attributes `sex` and `name`) which motivates the pattern matching notation:

```
Employee.allInstancesOf()
        ->select(Employee{salary=x,dept_id=5,... } |
                        x <> null and x>2000 )
```

Access to the implicit object id is forbidden; and we suggest to construct shadow-tuples completely, i.e. not producing *partial* tuples or the like which tends to complicate the type inference. The "..." notation is thus only used in patterns, not in declaration of tuples (introducing a concept like extensible records as in Isabelle).

8 Conclusion

The lively discussion both during the panel discussion as well as for each paper that was presented showed again that the OCL community is a very active community. Moreover, it showed that OCL, even though it is a mature language that is widely used, has still areas in which the language can be improved. We all will look forward to upcoming version of the OCL standard.

Acknowledgments. We would like to thank all participants of this years OCL workshop for their active contributions to the discussions at the workshop. This lively discussions are a significant contribution to the success of the OCL workshop series.

References

[1] Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Active Operations on Collections. In: Petriu, D., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 6394, pp. 91–105. Springer Berlin Heidelberg (2010)

[2] Brucker, A.D., Chiorean, D., Clark, T., Demuth, B., Gogolla, M., Plotnikov, D., Rumpe, B., Willink, E.D., Wolff, B.: Report on the Aachen OCL meeting. In: Cabot, J., Gogolla, M., Rath, I., Willink, E. (eds.) Proceedings of the MoDELs 2013 OCL Workshop (OCL 2013). CEUR Workshop Proceedings, vol. 1092, pp. 103–111. ceur-ws.org (2013)

[3] Brucker, A.D., Tuong, F., Wolff, B.: Featherweight ocl: A proposal for a machine-checked formal semantics for ocl 2.5. Archive of Formal Proofs (Jan 2014), http://afp.sf.net/entries/Featherweight_OCL.shtml, Formal proof development
[4] Clark, T., Warmer, J. (eds.): Object Modeling with the OCL, The Rationale behind the Object Constraint Language, Lecture Notes in Computer Science, vol. 2263. Springer (2002)
[5] Kosiuczenko, P.: Specification of invariability in ocl. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Model Driven Engineering Languages and Systems (MoDELs). Lecture Notes in Computer Science, vol. 4199, pp. 676–691. Springer-Verlag, Heidelberg (2006)
[6] Object constraint language specification (version 1.1) (Sep 1997), available as OMG document ad/97-08-08
[7] Object Management Group (OMG): Object Constraint Language (UML), version 2.3.1 (2012), http://www.omg.org/spec/OCL/
[8] (OMG), O.M.G.: Unified Modeling Language (UML), Version 1.5. http://www.omg.org/spec/UML/1.5/ (Mar 2003)
[9] (OMG), O.M.G.: Object Constraint Language (OCL), Version 2.4. http://www.omg.org/spec/OCL/2.4/ (Feb 2014)
[10] Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
[11] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edn. (2003)