Visual modelling of behaviour in data-intensive business applications

Piotr Habela[*], Anis Charfi[#], Axel Spriestersbach[#]

[*] Polish Japanese Institute of Information Technology
[#] SAP Research CEC Darmstadt

Abstract. Visual modelling has succeeded for example in the design of database schemas. But are there other areas in this domain where visual modelling may be useful? In this paper we describe experiences and results of the work on the design of a visual modelling language based on UML and Model Driven Architecture approach. The focus is on behaviour specification, which so far is rather handled through textual, platform-specific programming languages. We look for improving the productivity and the accessibility of this part of software development, in the context of a solution that is based on UML and more particularly on UML Actions.

1 Introduction

In the area of database application development, the term modelling is most frequently associated with the design of database schema. Indeed, this is the application where two criteria that seem critical for a successful software modelling solution - increased level of abstraction and approachable notation have been met. Various entity-relationship solutions offer intuitive visual syntax and some degree of platform-independence, at the same time being precise enough to allow automatic generation of database schema. One of the goals of the VIDE project at the platform-independent modelling level was to find out, what other aspects of business software development could benefit from a similar approach – that is – from abstract, visual specification that would allow generating target platform artifacts. The Object Management Group's initiative Model Driven Development (MDA) has basically the same objective, especially when considering its branch of work related with the concept of executable modelling. For these reasons, and because of the interests of the partners involved in the project, the development of a modelling solution based on OMG UML was assumed. Given the maturity of the structure modelling solutions in UML – class diagrams in particular – and still relatively weak adoption of precise behavioural modelling, the latter area became the primary subject of research.

Behavioural modelling constructs in previous versions of UML remained rather informal mean of communication, lacking the level of precision that would make them useful for generating code. UML 2 [OMG07] have been geared towards executable modelling through introducing fine-grained behavioural constructs - in Structured Activities and Actions units and a work on their executable semantics have been initi-

This work is supported by the EC 6-th FP, Project VIDE, IST 033606 STP

ated [OMG05]. UML 2 defines standardized actions, i.e., atomic behaviour modelling primitives for various purposes such as creating an object, calling a method, setting a variable or a structural feature, etc. However, UML does define standard concrete syntaxes. The lack of concrete syntaxes means that the users need to understand all details of the complex UML meta-model and they have to work directly with the meta-model concepts. This raises a question whether such syntax should be textual, visual or both. Another important question to be investigated is providing adequate means for expressions and queries, as the sole use of UML Actions would result with a too low level of abstraction (compared to e.g. database query languages). Hence, even in the layer of abstract syntax and underlying semantics that are already to some extent standardised, some open design questions remain. In this paper however, we are going to focus only on visualisation of the underlying behavioural constructs.

The rest of this paper is organised as follows. Section 2 summarizes the current shape of UML specification as the base of VIDE language development. In Section 3 we present an overview of the VIDE language. Section 4 and 5 provide the results and observations on visualising two areas of the behaviour respectively: UML actions (like updates, conditionals, loops) and expressions or queries. Section 6 concludes the paper

2 Visual notation of UML

Selection of the UML as the base for the programming language and modelling tool sets the context for the syntax as well, in the sense that some alignment should be achieved and certain analogies and metaphors exploited. The metamodel constructs of UML selected for VIDE do not have their concrete syntax specified, however – they are similar to or interact with UML constructs that do. The following conventions and analogies have been identified in the course of design:

- VIDE uses Structured Activities, hence it is desirable to visualise those constructs in a way similar to general activities known from activity diagrams.
- The ConditionalNode used by VIDE, although structured in different way, has a similar purpose as the DecisionNode whose standardised notation is also well know from activity diagrams.
- UML notation for instance (object) diagrams can be useful for representing expressions that return objects of a given type and involve navigation through links.

Apart from that, a question occurred, if arbitrary layout (positioning) of elements is desirable for Actions and Structured Activities visualisation. On one hand, such feature was natural for activity diagrams and the constructs used by VIDE can be considered just as an extension of those diagrams towards more precise behaviour specification. On the other hand, for such fine-grained constructs, this could obscure the view of code and make programming much slower. This argument is raised frequently against diagramming at this level, in favour of textual coding. To address these issues, it has been decided that VIDE should provide a complete textual syntax in addition to the visual one. Further, the visual notation was inspired in several regards by textual

notations. In addition, it has been decided that the layout of the visual and textual syntaxes should be maintained automatically. The motivation behind the automatic layout feature is twofold:

- Freeing the developer from manually positioning newly created constructs in order to make coding faster, and making the diagram, although visual, arranged in a way that would be natural from textual programmer.
- Providing a default visual layout for model elements that were created through textual coding and need to be visualised afterwards.

3 Scope and semantics of VIDE language

The abstract syntax of the VIDE language for PIM behaviour modelling is based on the UML meta-model and in particular the units of Structured Activities and UML Actions. Further, two concrete syntaxes were defined for this language: a visual syntax and a textual syntax. The abovementioned selection of behavioural constructs in the VIDE meta-model resulted in a rather traditional set of language constructs, resembling those found in a general purpose programming language. Hence the provision of a textual syntax for it was a rather obvious solution.

The choice of UML as the base of the language provides an object-oriented data model and brings the following specific design assumptions:

- Modelling of the structural parts of applications using class and package diagrams that constitute the most common and mature part of UML specification.
- Availability of UML extension mechanism involving profiles and stereotypes that can represent specific notions related with integration and may provide additional details essential for executable semantics.
- Connecting behaviour and structural models through operations inside class model that is setup familiar for developers used to platform-specific programming languages.
- Use of fine-grained behavioural constructs belonging to Activities unit, which potentially allows embedding the language inside activity or state diagrams.

An important observation is that the development of data-intensive applications demands data retrieval constructs of a higher level than the one offered by UML Actions. To address it and at the same time, to keep the solution standard-compliant, OMG OCL [OMG06] specification was adopted for the expression part of the language. Hence, the data read actions are replaced in VIDE by OCL expressions, which results with a programming language having expression constructs of expressive power analogous to the one of common query languages.

With the above mentioned considerations, the following elements of the language needed to be designed:

- Textual syntax for UML Actions and Structured Activities elements adopted by VIDE and the alignment of that with the standard textual syntax of OCL.

- A visual syntax for the abovementioned constructs that shields the users from the complexity of the meta-model and that avoids errors that are found in the textual languages such as bracketing and statement termination. Further, the notation should be loosely based on the style and metaphors of already standardised UML notation. And it should enable features in supporting editor such as contextual hints, auto-completion, auto-layout, etc.
- Visualising expressions in such a way that would mitigate the issues of OCL textual syntax – namely, its unconventional syntax and complexity of resulting expressions for certain kinds of queries (for example – in case of joins).

4 Visual Notation of UML Actions

The main assumptions behind the visual syntax and its realisation in VIDE prototype were the alignment with the standardised syntax of higher-level UML behavioural constructs, overcoming the main difficulties related with textual syntax, maximising the potential of the notation for providing contextual support and maintaining a possibly simple correspondence between notation elements and underlying metamodel constructs. Apart from that, the possibility of combining textual and visual syntax in a model of a single method was considered desirable to reduce the size of diagrams.

In the proposed visual notation actions are realized as templates, which contain placeholders for expressions that need to be set. These placeholders correspond to the action properties and/or to its pins. In addition, they are immediately visible in the notation and can be edited directly. As a result the users do not have to know all details of the UML meta-model that are relevant for a certain action. They only have to set values of the placeholders in the action template. Thus, the notation shields the user from the meta-model complexity.

We defined a visual notation for the most needed actions including object creation, method invocation, access and manipulation of object properties, as well as for the definition and manipulation of variables. In addition, the visual notation supports structured control flow nodes, which allow defining control flow.

The visual notation borrows several concepts from textual notations. First, the notation constrains the order of the actions to be sequential. Second, control flow is expressed implicitly via structured nodes. The same applies for data flow, which is expressed indirectly via variables. These decisions on the control flow and data flow keep the notation compact and prevent visual cluttering. Further, the similarity to textual languages allows exploiting the textual readership skills of users and also enable automatic layout.

The figure below shows the representation of three structured control flow nodes in the visual notation: sequence node, conditional node, and expansion region. For the expansion region, which is used for iteration the template has three place holders: the name of an iterator variable *<Item Name>*, the name of a collection *<Collection>*,

and the actions to be repeated for each element in the collection (the sequence node in the middle).

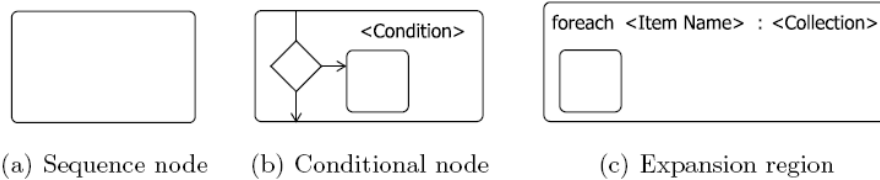

Fig. 1. Visual syntax for Structured Activities

The figure below shows the representation of three basic actions: *callOperationAction* (on the left), *createObjectAction* (in the middle), and *replyAction* (on the right). For the *callOperationAction* the template takes at least two place holders: the target object of the call and the operation to be called. If the operation takes input parameters then parameter text fields will be added dynamically in the lower compartment of the notation. For the *createObjectAction* a classifier has to be specified and also the name of a temporary variable that holds the created object. For the *replyAction* only one place holder expression with the value to be returned has to be specified.

Fig. 2. Visual syntax for selected Actions

The figure below shows the representation of the *AddVariableValueAction*, which allows a value to be inserted into or assigned to a variable. Two place holders have to be specified: the variable name and a value expression. In addition, if the specified variable is multi-valued, the value will be added to the variable, otherwise the existing value will be removed before adding the new value. In this case, the action would result in a simple assignment. If the specified variable is ordered (and multi-valued), an additional text field appears in the visual notation of the action where another placeholder has to be set: the insert position. The figure below shows also the *removeVariableValueAction*, which is the counter part to *AddVariableValueAction*.

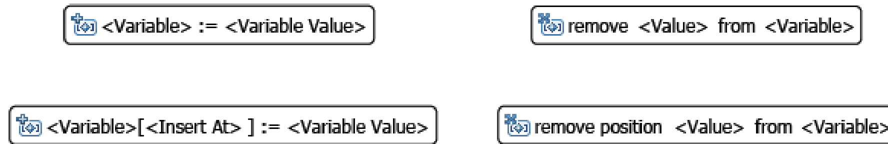

Fig. 3. Visual syntax for Variable-related Actions

The figure below shows the visual notation of structural feature actions. These actions manipulate the values of structural features of objects, e.g., properties and association ends. These actions are very similar to variable actions.

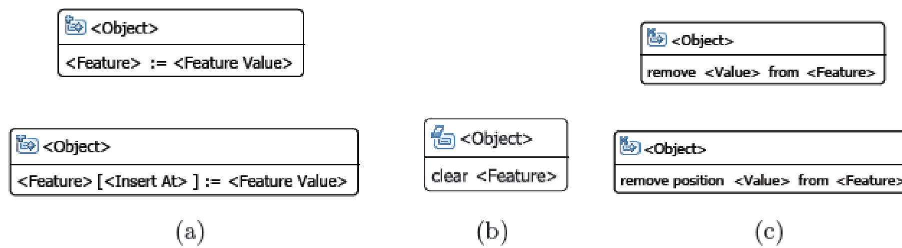

Fig. 4. Visual syntax for Structural Feature-related Actions

The *AddStructuralFeatureValueAction* (shown in (a)) adds a value to a structural feature of an object. Exactly like an *AddVariableValueAction*, this action inserts or assigns a value to the structural feature. Two placeholders have to be specified in this action template: the target object and the structural feature. The upper notation is used for simple assignments and for multi-valued unordered structural features (i.e., in case no insertion position must be specified). The lower notation is used when the insertion position must be specified, i.e., in case of multi-valued and ordered structural features. The visual representation of the *clearFeatureValueAction* is shown in the middle (b) whereas the representation of the *removeStructuralFeatureValueAction* is shown on the right (c). For the *removeStructuralFeatureValueAction* the object and one of its structural features have to be specified and a position may have to be specified in case of a multi-valued and unordered feature.

The arrangement of the VIDE textual syntax is to some extent suitable to follow in the visual notation and the respective interaction in the editor. However, the main difference here is that the creation of a statement is realised a more like from "the middle". That is, instead of specifying the (left hand) expression first and later choosing the instruction to apply on it, the sequence of steps is as follows.

1. First, the developer chooses appropriate instruction (e.g. ExpansionRegion for iteration or AddStructuralFeatureValueAction for object's attribute update).
2. A visual symbol with a template for respective parameters of the instruction is shown.
3. Now, to get most from the contextual hints, the developer should start from specifying the parameter that represents the subject of the instruction (for example the

expression returning the objects whose attribute we are going to update). When writing the expression the developer can be assisted based on the names visible in the environment of the method (and the local environments of particular subexpressions).

4. When the above expression is finished and validated, the type of the expression result is determined and this information can be used so as to support the specification of the remaining parameters of the statement. For example, a list of attribute names in update statement can be filled with the names of attributes available in respective class.
5. The next parameter of the expression provides further type information that can be used to narrow the hints for the remaining parameters.

Consider the example in the figure below. It represents a body of a User.observe(a : Auction) method for a simple auction site system (the complete class diagram for the sample is included in the appendix).

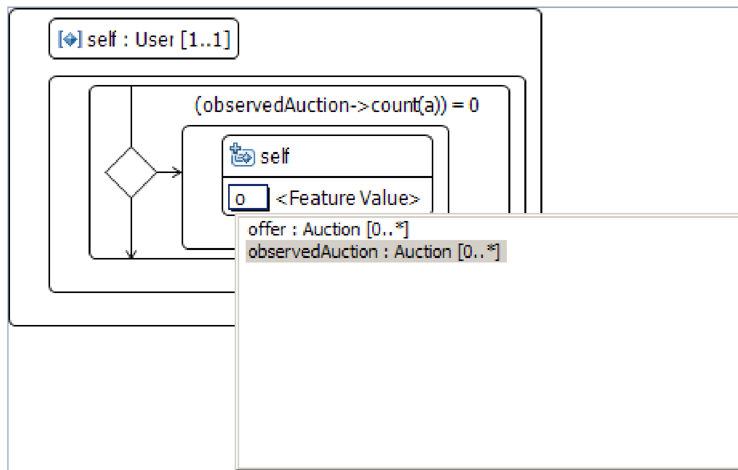

Fig. 5. Contextual hints using the information from the metamodel instance

Here, inside the conditional block an AddStructuralFeatureValueAction element has been added. The object to be modified in this statement was determined by writing a simple expression consisting of the **self** keyword. Now, the type of the object being modified is determined (User class) and the structural features of the respective class are shown as the hint. Now, the type of the attribute being modified is known. Hence, the list of hints for the right hand of the assignment, that would normally contain all names visible inside the method environment, can be narrowed to include only the ones of the matching type (which is Auction).

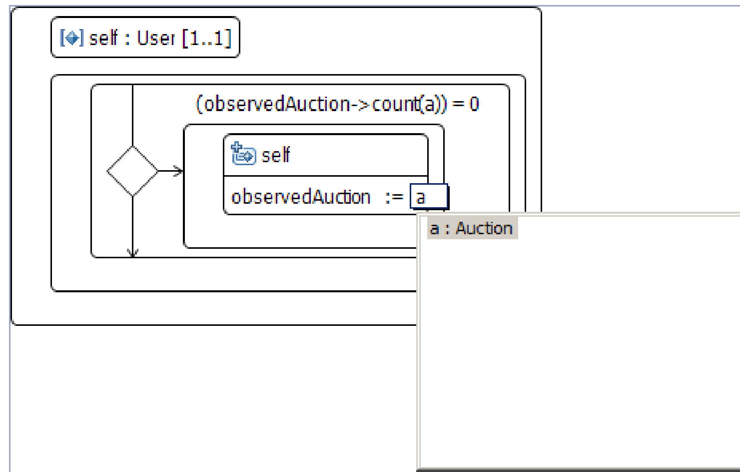

Fig. 6. Contextual hints narrowed using the information from other parameter of an action

Use of Structural Activities in the underlying metamodel reduced the need for control and object flows. Particularly, the sequence of statements is determined by the SequenceNode element. Hence the notation differs from the usual UML activity diagram with that instead of connecting activities with flow arrows, the sequence is determined by their automatic top-down arrangement inside containing SequenceNode elements. This solution is motivated by speeding up the process of coding and avoiding readability issues that could result from an arbitrary, non sequential layout of multiple fine-grained elements. A more complex sample of visual code (without visualisation of expression) is shown in the figure below.

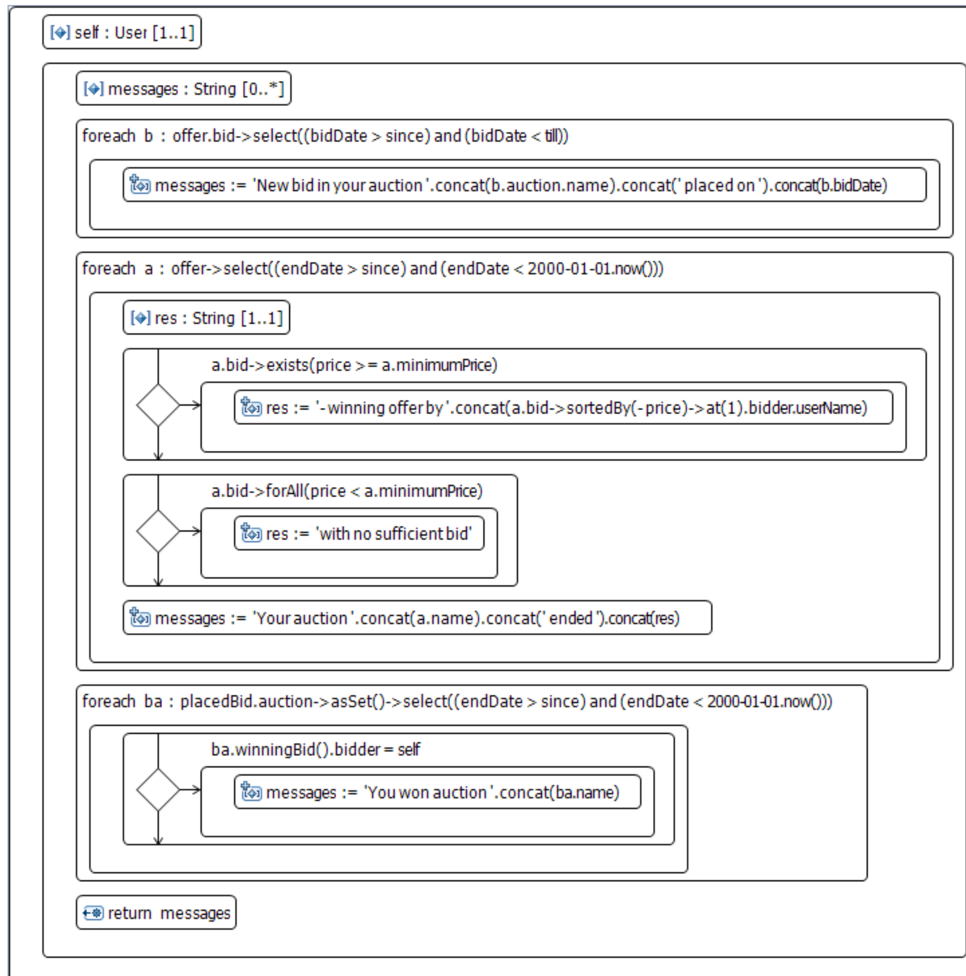

Fig. 7. Example of a visualized method body (expressions remain textual here)

Another interesting concept intended to make the notation more familiar to UML modellers is exploiting some analogies to UML instance and class diagrams. In fact, as it will be explained, this concept is especially useful for expressions, but can be also applied to imperative constructs. Namely, in VIDE visual notation the attribute updating constructs (see the figure above) have been designed so as to resemble object symbol from instance diagrams – where the expression retrieving the object is located in the top field (where the name of object and it class would occur in instance diagrams), while the attribute name and its value to be assigned resembles the attribute value representation of instance diagrams or attribute default value assignment on class diagrams. A similar arrangement has been also applied for CallOperationAction – see a figure below. In this case the parameters of the operation called are represented in similar way as attributes in instance diagrams.

```
user->one(userName = uName)
newAuction := createAuction
aName = aName
aDesc = aDesc
cPrice = cPrice
minPrice = mPrice
eTime = eTime
cat = category->select(categoryName = cName)
rLimit = 0
```

Fig. 8. Visual syntax for Call Operation Action

As can be seen from the above samples, the main challenge for making the language more approachable is finding an optimum way of visualising expressions. This will be discussed in the next subsection.

Representing expressions

One of the primary concerns in design of the visual notation was representing expressions. Their textual counterpart having standardised OCL syntax is the primary source of VIDE textual code complexity. Combination of several iterator operations, especially involving joins and nested queries inside predicates can result in rather obscure code. The considerations for overcoming this are presented in the next section. Here, we focus on how the expression part can be combined with the remaining part of the language in the design of the visual notation for imperative constructs.

The design of the visual notation was to some extent influenced by the textual syntax, while the latter was designed to comply with the syntax of OCL that constitutes a major part of the language. The usual pattern in OCL syntax is subexpression-operation-parameters. This is visible e.g. in iterator operations:

```
employee->select(job='programmer')->forAll(salary>1000)
```

and it also results in a postfix style syntax for collection operations:

```
order.items.price->sum()
```

Only few cases, when this style of syntax would be especially exotic constitute an exception and are realised in a more conventional way. An example can be traditional (prefix style) syntax for not operator and the syntax for the IfExp:

```
if (not discount) then price else price*0.9 endif
```

Similar style was followed by textual VIDE syntax for imperative constructs like loops, iteration, link creation or feature value addition. For example:

```
product->select(not available) foreach { p |
          p.reorder(100);
}
```

As the assumed area of application of the VIDE language are data intensive applications, complex expressions may need to be used. Hence the challenge for visual syntax is to remove part of the complexity that occurs for OCL expressions. Two approaches were originally considered. One of them was aimed at providing graphical parentheses, so that the (potentially nested) environments for particular subexpressions can be shown. The natural sequence of creating such blocks would allow to collect type information so as to provide hints for the construction of subsequent steps. However, it was concluded that such syntax would be too space-consuming and in addition this would not match the intuitions based on the existing elements of UML syntax. Since it was impossible to combine such approach with the UML instance diagram-like notation, this approach has been given up in favour of the Query By Example (QBE) based approach.

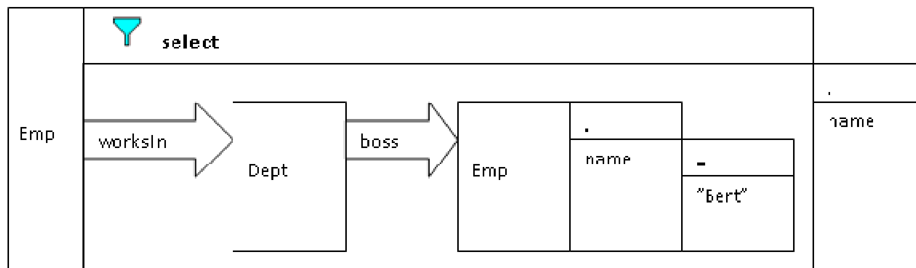

Fig. 9. Syntax considered initially to visualize nested scopes inside iterators

The language design, called Object Query By Example (OQBE) went beyond just a concrete syntax for UML / OCL metamodel, providing instead new, dedicated constructs based on the notion of example of an object. This includes the constructs like object example, link example and attribute example. For representing examples the UML instance diagram syntax was adopted. As the OQBE constitutes a separate language over OCL, we devote more space to this topic. However, the code generation algorithm itself will be the subject of another paper.

Object examples connected with link examples can represent respective data structure examples. If more than one mutually unconnected parts of example diagram exist, they are considered to represent a Cartesian product of their results. Attribute example may play different roles:

- **Predicate** – if the attribute itself or the result of operation applied to it (including various kinds of comparison) provides a Boolean value and no specific flag is attached to the attribute example.
- **Output element** – if an attribute example has the *output* flag attached.
- **Sorting criterion** – if the *sort* flag with needed properties (priority, order) is attached (see the figure below for example of its usage).

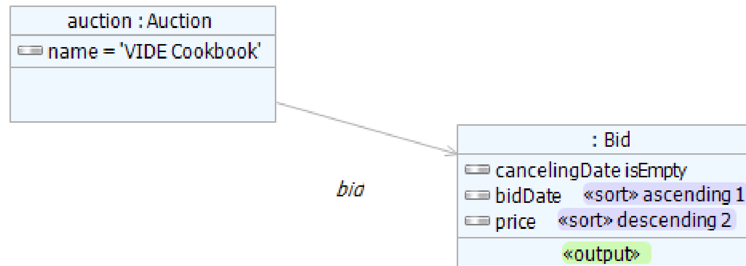

Fig. 10. OQBE expression sample involving selection, navigation, sorting and projection ("get non-cancelled bids for 'VIDE Cookbook' and order them according to date and price")

The *output* flag can be applied to object example as a whole (in that case it marks a whole object to constitute an element of the expression result) or to an attribute. More than one *output* flags can be used in a single expression. In that case however, the result is assumed to be of a Tuple type and hence the particular flags need to be accompanied with appropriate tuple field names as shown in the figure below.

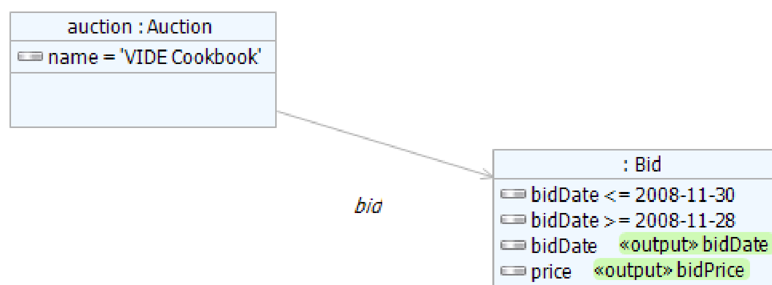

Fig. 11. OQBE expression with a tuple result consisting of two named fields ("get dates and prices of bids placed for 'VIDE cookbook' between the given dates")

Since using single names and operations on them can be too limiting for some of the abovementioned applications of attribute example, the language provides a similar construct where instead of attribute name an arbitrary OCL code can be embedded (see figure below that shows this construct used for operation invocation).

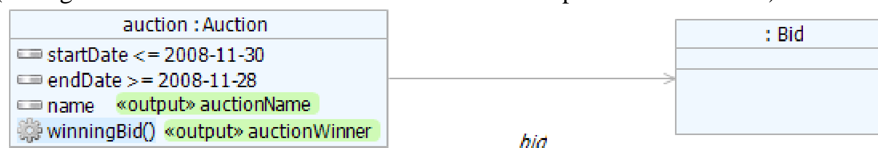

Fig. 12. OQBE expression including an embedded OCL code ("get names and winning bids for auctions active in the period specified")

This feature allowed to overcome some technical limitations of the prototype tool built (as in case of the above sample), however it can be also useful as a shortcut or

when an expression is too complex to be fully visualized (for example – multiplication or adding several values stored in attributes of the same of a different object and returning them as the expression result).

Although the readability considerations limit the usage of connections between attribute examples, a simple construct called *Comparator* was introduced in order to represent comparisons between attribute or object examples. Its usage is shown in the figure below.

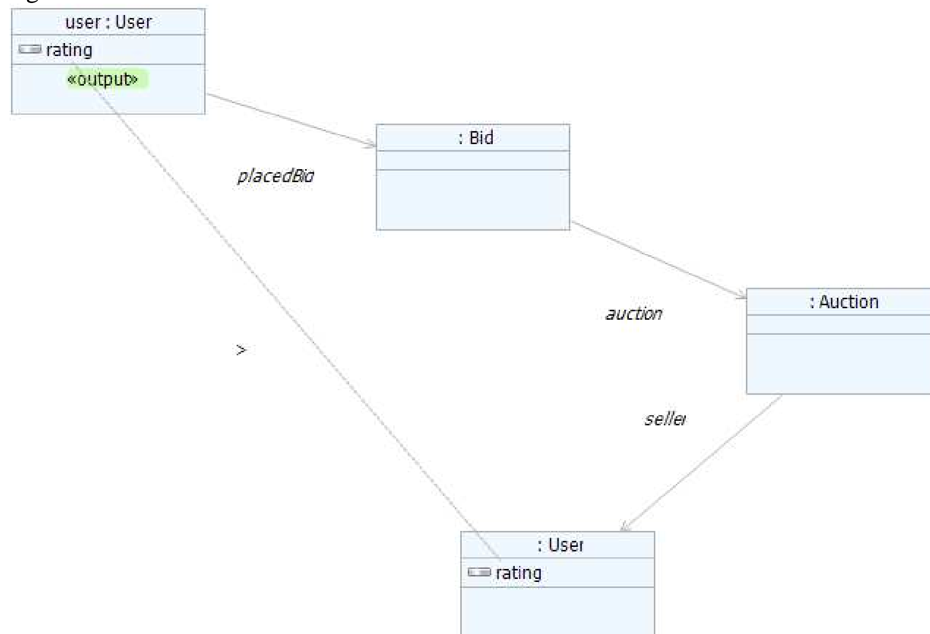

Fig. 13. OQBE expression involving the comparator construct ("get users who used to bid for items sold by users with a lower rating")

One of the features that make OQBE different from traditional QBE solutions based on relational model is the availability of nested structures in expression result construction. In textual OCL this is realized by nesting subexpressions inside explicit Tuple constructor. In the OQBE the tuple is constructed implicitly by providing name to the *output* flags. Hence, the only element that was necessary to achieve nested result structures was a visual region inside which respective part of the diagram (containing its own output flags) could be nested. Note that the nested output region requires providing the field name for it.

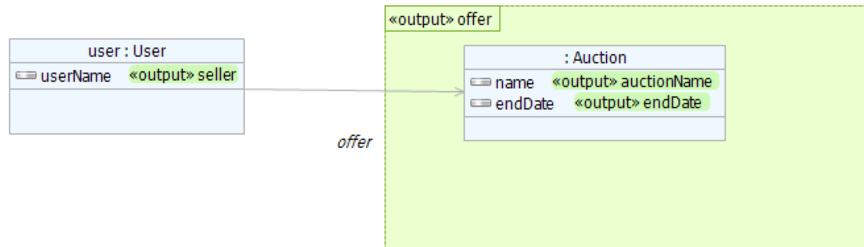

Fig. 14. OQBE expression with a nested structure result ("get user names together with sets of names and end dates of their auctions")

A similar visual construct was introduced for the general quantifier (*forAll*). Inside that region only navigation and predicates are allowed. From the example graph outside the *forAll* region that is connected with the examples inside the region only those instances will be included in the result for whose all navigations through the link passing the region boundary the condition is met. Hence a single use of the *forAll* region works as a `->select` + `->forAll` combination. On the other hand, in case the *forAll* is used inside another *forAll*, it is expected to return logical value rather and hence works as a sole `->forAll` operation.

Fig. 15. OQBE expression including a general quantifier ("return users who sell the 'Book' category items exclusively")

When experimenting with the language on some exemplary models we have found out that the most significant advantage of the OQBE over the textual syntax occurs in case of more complex queries that involves joins and / or complex conditions involving different objects. For example a "fraud detection" query that looked for pairs of users who used to bid in each other's auctions and at least one of them refused the purchase afterwards, requires the following textual code in OCL.

```
user->collect(u | user->collect(v |
Tuple{u1 = u, u2 = v}))

->select(u1<>u2)

->select(u1.placedBid->exists(b | (b.auction.seller=u2)
and (b.auction.winningBid()=b) and (b.auction.result-
>exists(refused))))

->select(u2.placedBid->exists(y | y.auction.seller=u1
))
```

```
->collect(
Tuple{user1 = u1.userName, user2 = u2.userName});
```

The same task could be realized with a full contextual hint assistance by building the following OQBE diagram.

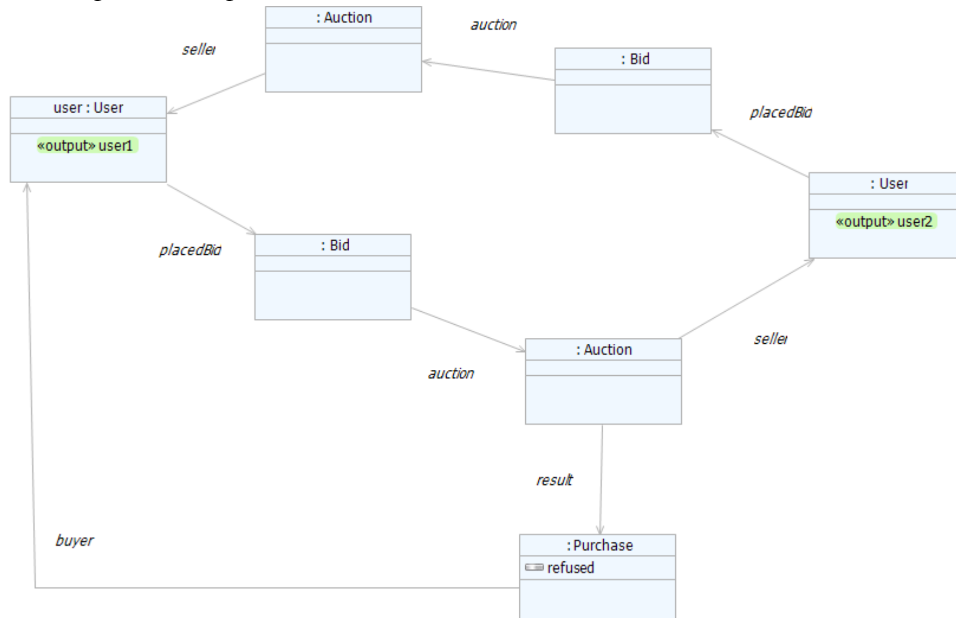

Fig. 16. OQBE expression involving intensive navigation in objects structure ("get pairs of users who placed bids in each other's auctions and at least one of them refused to buy")

Apart from the constructs described above several further features have been proposed in the course of research but not implemented in the prototype. One of them is a construct for invoking an operation on a collection (for example `->sum()` or `->asSet()`). This is to be realized in a similar way as e.g. the nested output region described above – the operation would be applied to the result produced by the example enclosed by this *aggregateOutput* region in regular way. Another postulate is analyzing the names connected to the output flags in case a diagram consists of two or more separate subgraphs and for those that are identical constructing union rather than Cartesian product of their results.

Conclusions

In this paper an overview of VIDE solutions related with visual UML-based programming on the platform independent level has been provided. The language under consideration is intended to be used on the frontier between the domain of modelling (where different graphical notations are common) and traditional programming (which is dominated by textual language). The solution proposed is a visual syntax

that accompanies and can be combined with a textual syntax with both syntax being mapped to the same UML and OCL metamodel constructs. With these syntaxes and the supporting editors three problems of behaviour modelling with UML Actions are solved: the lack of a concrete syntax, the complexity of the meta-model, and the insufficient tool support. By solving these problems we pave the way to a more widespread adoption of UML actions for behaviour modelling and provide tools that ease creating and manipulating action models.

While the fine granularity of the behaviour under consideration requires assuring high speed of coding and hence implies some similarities with textual syntax, additional expressiveness and metaphors has been sought for VIDE visual notation. The analogies with other mature part of UML followed involve activity and instance diagram symbols.

In the area of imperative statements, the differences between textual and visual VIDE language lie practically only in the layer of concrete syntax. However, the visual syntax and underlying editor functionality make it simpler to assure coding-time validation and contextual hints. Appropriate sequence of specifying a visual statement – encouraged by the syntax design – allows to collect type information and hence maximise contextual assistance.

For the expression part of the language a more sophisticated approach was followed, in order to fully exploit the metaphor derived from UML instance diagram (identified as the most promising concept), which resulted in an object-oriented realisation of the Query By Example paradigm. Hence the design incorporated a dedicated metamodel and syntax for a visual language named "Object Query By Example" (OQBE). As the approach involves a higher level of abstraction than regular OCL queries, there is no bidirectional mapping (only OQBE to OCL) and the respective OCL generation is performed in a transparent way. While for very simple queries a textual expression may be faster to write, the advantage of the OQBE becomes very visible in case of queries that include longer navigation paths, joins or comparisons. Not all kinds of queries can be fully visualised – some expressions, e.g. involving calculating an output value from several different attributes could simply become unreadable. For this reason it was made possible to include OCL textual code snippets onto the OQBE diagrams.

References

[OMG05] Object Management Group: Semantics of a Foundational Subset for Executable UML Models. Request For Proposal. ad/2005-04-02.

[OMG06] Object Management Group: Object Constraint Language version 2.0, May 2006. `www.omg.org/cgi-bin/doc?formal/2006-05-01`

[OMG07] Object Management Group: Unified Modeling Language: Superstructure version 2.1.1, February 2007. `www.omg.org/cgi-bin/doc?formal/2007-02-05`

Appendix

UML class model describing the schema used by visual syntax examples.

User
+userName : String
+realName : String
+password : String
+email : String
+rating : Integer
+observe (a : Auction)
+createAuction (aName : String, aDesc : String, cPrice : Integer, minPrice : Integer, eTime : Date, cat : Category, rLimit : Integer)
+getNotifications (since : Date, till : Date) : String { unique }
+getNewSubscribedCategoriesItems (since : Date) : Auction { unique }

Purchase
+refused : Boolean
+settle ()
+leaveSellerFeedback (rt : Integer, com : String) : Boolean
+leaveBuyerFeedback (rt : Integer, com : String) : Boolean

Auction
+name : String
+description : String
+ratingLimit : Integer
+calledPrice : Integer
+minimumPrice : Integer
+startDate : Date
+endDate : Date
+bid (biddingUser : User, maxPrice : Integer...
+winningBid () : Bid
+prolong (till : Date)
+cancel () : Boolean
+calculateCharge () : Real
+resolve ()
+sellersOtherItems () : Auction { unique }

Feedback
+rating : String
+comment : String

Bid
+date : Date
+price : Integer
+cancelingDate : Date[0..1]
+cancelingReason : String[0..1]
+cancel (reason : String)

Category
+categoryName : String

<<module>>
AuctionSite
+user : User[*] { unique }
+category : Category[*] { unique }
+auction : Auction[*] { unique }
+registerUser (uName : String, rName : String, pass : String, mail : String) : Boolean
+setAuction (cName : String, aName : String, aDesc : String, cPrice : Integer, mPrice : Integer, eTime : Date, uName : String) : Boolean
+WRITE_SAMPLE_DATA ()

Associations:
- User (1, +buyer) — Purchase (*, +won { unique })
- User (1, +seller) — Auction (*, +offer { unique })
- Purchase (*, +result { unique }) — Auction (1, +offer)
- Purchase (1, +sfSubject) — Feedback (0..1, +sellerFeedback)
- Purchase (1, +bfSubject) — Feedback (0..1, +buyerFeedback)
- User (1, +bidder) — Bid (*, +placedBid { unique })
- User (*, +observer { unique }) — Auction (*, +observedAuction { unique })
- Auction (1, +auction) — Bid (*, +bid { unique })
- Auction (*, +categoryItem { unique }) — Category (1, +auctionCategory)
- User (*, +subscriber { unique }) — Category (*, +subscribedCategory { unique })

This work is supported by the EC 6-th FP, Project VIDE, IST 033606 STP