Defining OCL expressions using templates

James S. Willans[1], Paul Sammut[1], Girish Maskeri[1],
Andy Evans[1] and Tony Clark[2]

[1]Department of Computer Science, University of York,
York, England, YO10 5DD
{jwillans|pauls|girishmr|andye}@cs.york.ac.uk
[2]Department of Computer Science, Kings College,
London, England, WC2R 2LS
anclark@dcs.kcl.ac.uk

Abstract. OCL expressions are an essential part of UML. The current versions of OCL fail to have a meta-model which means that the integration of OCL with the UML meta-model cannot be formally defined [1]. This can result in ambiguous descriptions of systems which may compromise designs. The need to redesign the OCL has been addressed by a number of proposals submitted to the OMG. In this paper we demonstrate how a definition for OCL can be stamped out from a small number of templates. Such an approach enables a high level of reuse and an increased confidence that the definition is correct. This work forms part of the 2U consortium's efforts for the definition of UML 2.0.

1. Introduction

It is useful to be able to express computational systems by the precise behaviour they should exhibit. For these we take the imperative approach typified by conventional programming languages. However, often it is desirable to describe systems not by their precise behaviour, but by declarative rules that the behaviour should conform to. In these cases it can be convenient to express systems using more abstract descriptions. With UML, such expressions are described using the Object Constraint Language (OCL).

The OCL has been part of the UML since its inception. However, the current versions of OCL fails to have a meta-model which means that the integration of OCL with the UML meta-model cannot be formally defined [1]. This can result in ambiguous descriptions of systems which may compromise designs. In order to address this a number of proposals have been submitted to the Object Management Group (OMG) to redesign OCL such that it has an underlying meta-model (see [2] for more details).

In this paper we describe how we have taken a template approach to defining OCL within the 2U submission for UML 2.0 [3] (the definition we arrive at follows closely that of [1]). Template oriented definitions are advantageous in that a high level of reuse is promoted. Critically, the OCL definition has been implemented in our meta-

modelling tool (MMT) [4] and we have built models using the definition, increasing our confidence of its correctness. A further contribution of this paper is to briefly illustrate how our definition of OCL expressions can be generalised to computational expressions (e.g. arithmetic operators), and integrated with an action definition in order to describe computations.

2. Background

The work described in this paper forms part of the 2U Consortiums efforts to define a submission for UML 2.0 [5]. The approach taken by the group is characterised by a number of strategies. In this section these are described.

2.1 Unambigious yet understandable

A definition of UML must be precise so that there is no ambiguity about what models built using the language mean. This involves separating those aspects of the definition that deal with representation (abstract syntax) with those that deal with meaning (semantics) [6]. The traditional approach to ensuring precision in languages such as UML is to define their meaning using formal (mathematical) abstractions (for example in [7] Statecharts are defined using Z [8]). While there is no question that these approaches offer the required level of precision, their highly abstract mathematical nature also makes them difficult to interpret.

The approach adopted by 2U is to model the syntax and semantics of UML 2.0 using precise UML class diagrams which are augmented with OCL constraints. This reflexive approach to language engineering is a powerful means of defining more complex languages from simpler ones. The precision of class diagrams means that they are unambiguous and the visual nature of the diagrams, and their wide spread adoption, enables these to be easily interpreted. The 2U approach uses both a visual and textual version of constrained class diagrams, the textual version is understood by MMT (discussed in section 2.3).

2.2 Promotion of reuse

Reuse is a core strategy for designing and building software. Within that context, the focus has been on how abstractions can be reused using mechanisms such as objects and inheritance. A weakness of this style of approach is that complete solutions are reused, and often it is the case that it is the structure of a solution rather than the details that have a high level of reusability.

The 2U approach has identified that much of the definition for UML 2.0 can be constructed from a small number of recurring structures (often referred to as patterns [9]). For instance, a commonly found structure is the container relationship where one element (conceptually) contains another. These reusable structures are encapsulated into templates which can be instantiated with data abstractions. The class diagram for the container template is shown in figure 1 (a). Templates are

instantiated by substituting the place-holders (enclosed by << >>). The template of figure 1 (a) might be instantiated using *Class_* and *Attribute_* parameters to define that a single class contains many attributes[1]. This is illustrated in figure 1 (b).

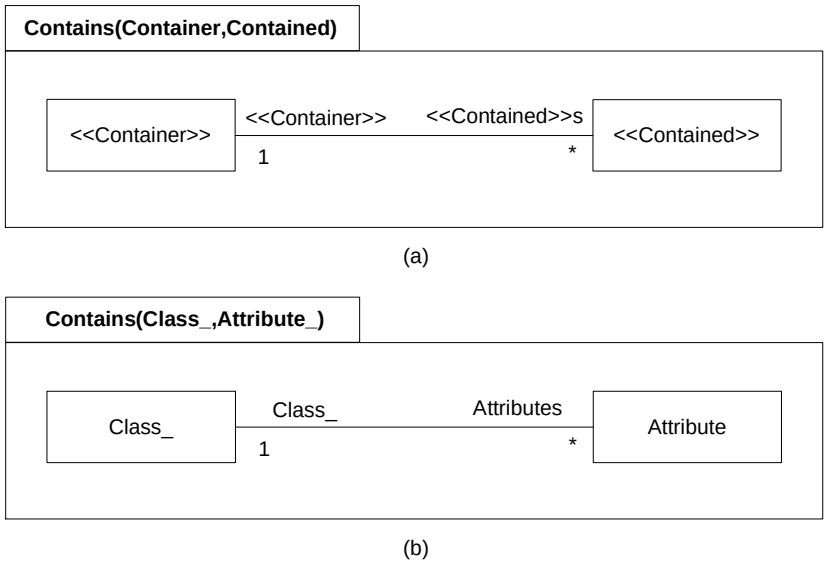

Fig. 1. A template describing the contains relationship

2.3 Correctness

An unambiguous and understandable definition ensures that the definition can be accurately interpreted with ease. The promotion of reuse using templates enables the definition to be rapidly constructed from components. However, neither of these strategies ensures that a definition meets its requirements. The only way this can be achieved reliably is by extensive testing. In the case of UML, this involves building models using the definition to determine the strengths and weaknesses of the definition in view of the requirements.

The meta-modelling tool (MMT) has been designed and constructed to support the 2U consortiums definition and testing of UML 2.0. MMT is a virtual machine that understands the textual version of constrained class diagrams and the construction of languages using templates. MMT supports the testing of language definitions at a number of levels. At a simple level it is able to check the definition to ensure it is syntactically correct (the importance of this in a definition the size of UML 2 should not be underestimated). MMT is also able to check that constraints hold within the definition to ensure that models are well formed. Most importantly, MMT is reflexive

[1] the underscore is used when naming abstractions to avoid conflict with the pre defined abstractions of MMT

which enables the building of new languages described using existing languages. Consequently, MMT can be used to build UML 2 models and check that our understanding of UML2 (encapsulated in the definition) is correctly defined.

3. OCL Definition

Part of a typical OCL expression may look like the following:

```
bank.hasMoney and bank.hasStaff
```

This expression specifies that the *and* statement is true if both the slot *hasMoney* owned by *bank* and the slot *hasStaff* also owned by *bank* are true. This example illustrates two fundamental characteristics of expressions:

1. Expressions can contain expressions as operands. In the case of the above example the *and* expression has a dot expressions as its left and right operands (similarly, the dot expressions themselves have two operands). This means, that specific expressions (e.g. *and*) must be generalised from some common abstract expression type in order to support polymorphism.

2. Expressions have a type which they evaluate to. In the above example the *and* expression evaluates to a boolean type. Associated with the type is a value, in the case of a boolean expression this value is true or false.

This essence of expressions is captured in the template illustrated in figure 2. In this the syntax of a concrete expression is specialized from an abstract expression and has a type. The semantic domain specifies that a concrete expression evaluation is generalized from an abstract expression and has a value which it evaluates to. The semantic mapping characterises that a concrete expression can have many evaluation instances.

The value of an expression evaluation should be valid in view of its type. For instance, a boolean expression should only evaluate to true or false. The template shown in figure 2 is therefore augmented with the following well formedness constraint:

```
context Expression::SemanticMapping::<<Evaluation>>
inv: <<Evaluation>>ValueCommutes
  self.of_.type = self.value.of_
  fail <<Expression>>+”Evaluation value failed to commute”
end
```

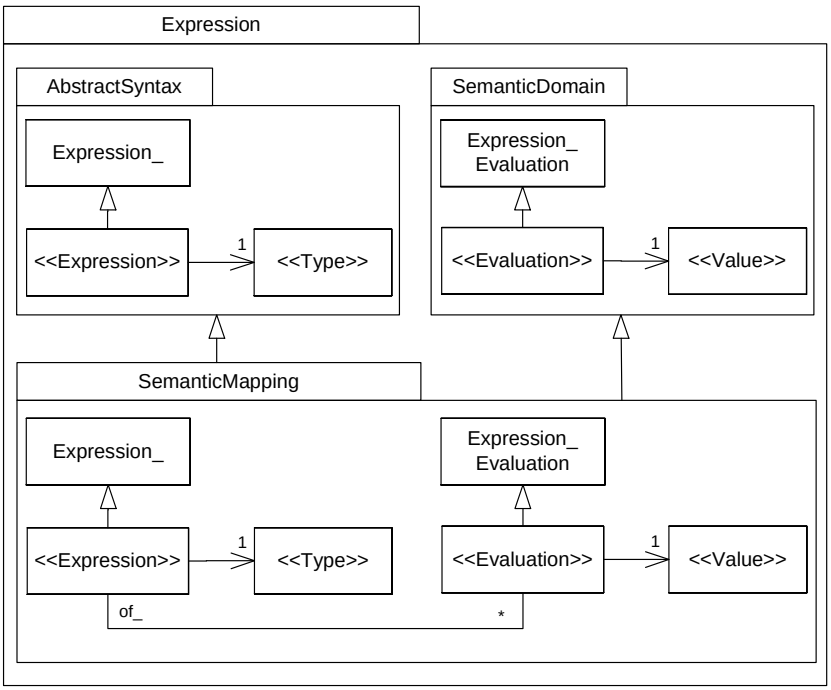

Fig. 2. Basic expressions template

Although the basic expressions template captures the essence of expressions it fails to specify how expressions can have operands which are themselves expressions. There are two broad classes of expressions, those with one operand (unary expressions) and those with two operands (binary expressions). The three template of figure 2 (abstract syntax, semantic domain and semantic mapping) can be used as a basis for deriving (stamping out) further templates which deal with the respective domain for each class of expressions (unary and binary). Figure 3 shows a binary expressions semantic mapping templates which is achieved by adding two operands to the result of stamping out figure 2. Note that from now on we will show only the semantic mapping of templates and definitions for brevity of presentation.

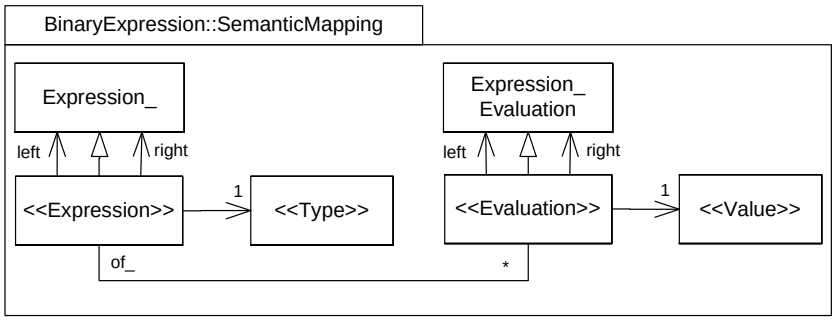

Fig. 3. Binary expression template

Given the binary and unary expressions templates, we are now in a position to be able to stamp out concrete expressions. For instance, illustrated in figure 4 is an *and* expression, with a boolean type and value, which is stamped out using the template of figure 3. Within an *and* expression, both operands should be also of type boolean:

```
context AndExpression::AbstractSyntax::And_
inv: operandsAreBoolean
  self.left.type.isKindOf(AbstractSyntax::Boolean_) and
  self.right.type.isKindOf(AbstractSyntax::Boolean_)
  fail: "And_ operands should be of type Boolean_"
end
```

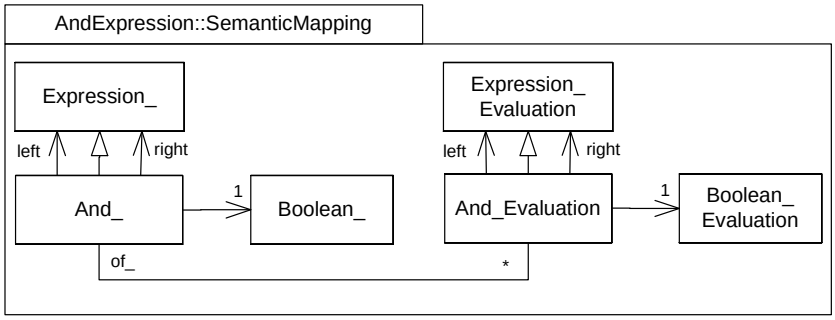

Fig. 4. *And* expression

The result of evaluating an *and* expression (its value) should be the conjunction of its two operands:

```
context AndExpression::SemanticMapping::And_Evaluation
inv: isAndOfLeftAndRightOperands
  self.value = self.left.value and self.right.value
  fail: "Value of And_ evaluation is not conjunction of its
        operands"
end
```

Using the unary and binary expression templates we are able to define a number of OCL expression constructs in the same manner as *and*. When this definition is used to describe non-trivial OCL constraints, the resulting instance model will visually look like a tree of expressions and sub-expressions (akin to programming language expression trees). At the bottom level of expression trees are values, variables holding values or query methods that return values (we collectively refer to these as variable references). We have not yet defined how these are described within our definition of OCL.

In the example at the beginning of this section, examples of variable references are *bank.hasMoney* and *bank.hasStaff*. When referring to variables that exist in the same object as the expression, the dot expression is omitted and the statement is more simply cast as *hasMoney* and *hasStaff*. However, since this is a short hand for *self.hasMoney* and *self.hasStaff*, variable references can be seen as always residing on the right hand side of a dot expression. We call the right hand side of a dot expression a property call. A property call has a reference to abstractions which are of type property (variables, query methods and association ends). The left hand side of a dot expression may be another dot expression or a special variable called *self* which binds every object to itself. The abstract syntax of variable expression and its relation to the other part of the OCL definition is shown in figure 5. To exemplify this definition, an instance model of the following expression:

```
self.dog.cat.mouse
```

is shown in figure 6.

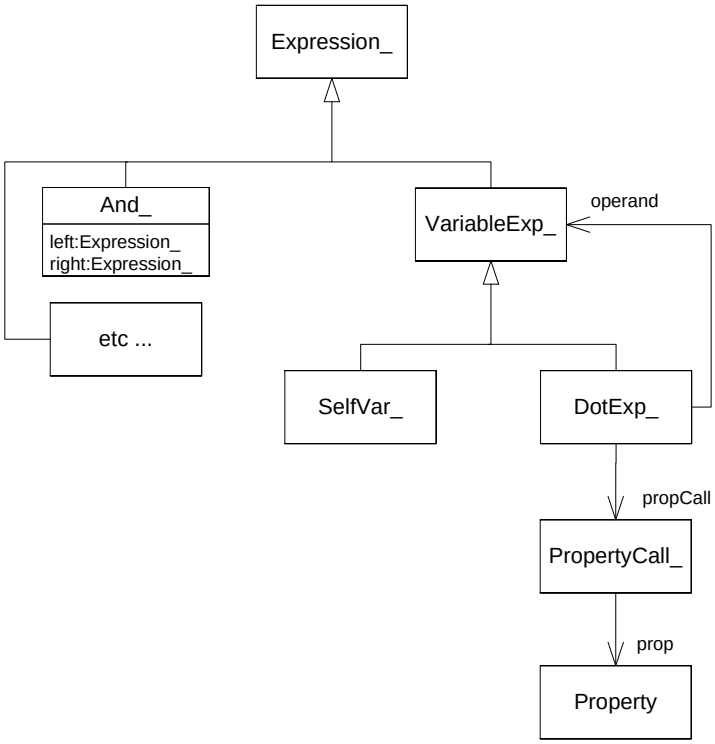

Fig. 5. Abstract syntax of template defined OCL expressions

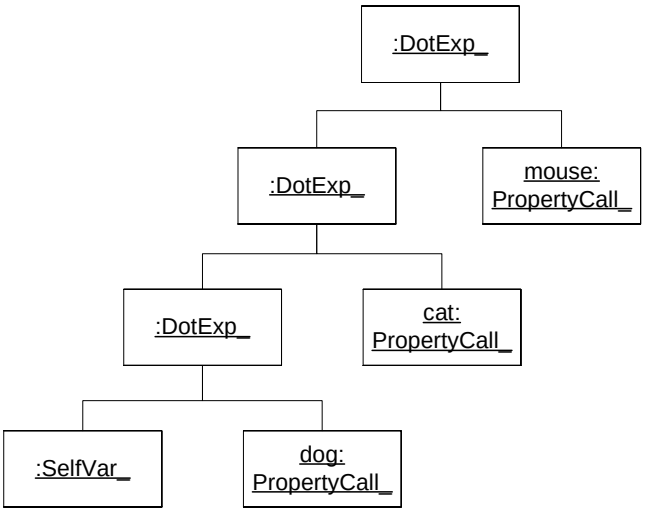

Fig. 6. `self.dog.cat.mouse` variable expression

An OCL invariant must always evaluate to true. At the top level of an OCL expression is an invariant abstraction (*Invariant_*) that is not an expression but contains an expression. Many invariants can be contained by a class (*Class_*) as described in the syntax definition of figure 7. An invariant's expression must always be of boolean type:

```
context Invariant::SemanticMapping::Invariant_
inv: expressionIsBoolean
  self.rootExp.type.isKindOf(AbstractSyntax::Boolean_)
  fail: "An invariant's expression must be of Boolean_ type"
```

and evaluate to true (i.e. the constraint must always hold):

```
context Invariant::SemanticMapping::Invariant_Evaluation
inv: evaluatesBooleanTrue
  self.rootExp.value = true
  fail: "An invariant must evaluate to true"
end
```

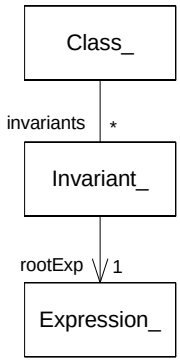

Fig. 7. A class contains many invariant and each invariant has an expression

4. Example

In the previous section we have described and illustrated our approach to defining OCL expressions. We have found that most of the OCL can be stamped out from the small number of templates described, however there are exceptions such as those expressions that deal with iterates (*forall* and *collect*, for example). These require a

little bit more work beyond the standard templates described in the previous section, however we refrain from detailing that here.

Our confidence in the templates is augmented by the fact we have built models using the stamped out OCL within MMT. To illustrate this, consider once again the constraint we described at the beginning of section 3:

```
bank.hasMoney and bank.hasStaff
```

A (syntax) instance model which includes this constraint is shown in figure 8. This describes that a *bank* class owns two attributes called *hasMoney* and *hasStaff*. The *bank* class also owns a constraint (*inv*) whose value must always be true. The constraint has an and expression (*andExp*) which has two sub-dot expressions called *dotX* and *dotY*. Each of these dot expressions has a right operand linked to a property call (*xcall*,*ycall*) these property calls link back to the *hasMoney* and the *hasStaff* attributes. The left operand of the dot expressions (*xcall*,*ycall*) are linked to the *self* variable *selfVar* which is linked back to the *bank* class.

5. Using expressions with actions

In the previous sections we have described our approach to defining OCL expressions and exemplified the definition with a small example. Within that discussion, we have focused on the use of OCL expressions in the context of invariants that specify some axiom. However, expressions are also used to describe how non-state changing (in terms of objects and slots) computations take place. For instance, the behaviour of arithmetic operators: `(2+(3*4))/2`. In that context it is important to understand the interaction between expressions and state changing computations that are described using actions. For example, the following model combines a slot update action (=) and an expression tree:

```
int x = (2+(3*4))/2
```

Our definition of the expression/action simply states that actions can contain expressions or actions as their sub-actions, but expressions can only contain further expressions. This is more concretely illustrated in figure 9. The expressions templates (figures 2 and 3) are augmented to include an abstract behaviour type (to enable polymorphism), from which the abstract expression type is generalized. This abstract behaviour type can be viewed as a plug in point for actions. The templates which are used to stamp out the action language (which we refrain from giving here but which are detailed in [10])[2] also have an abstract behaviour type. Actions are defined as having actions of type behaviour, and thus can be either expressions or actions. However, expressions are only able to have further expressions as their operands.

[2] However the architecture of figure 9 does allude to our definition of actions by indicating that actions are characterised by pre and post states in addition to the value of their computations (as with expressions).

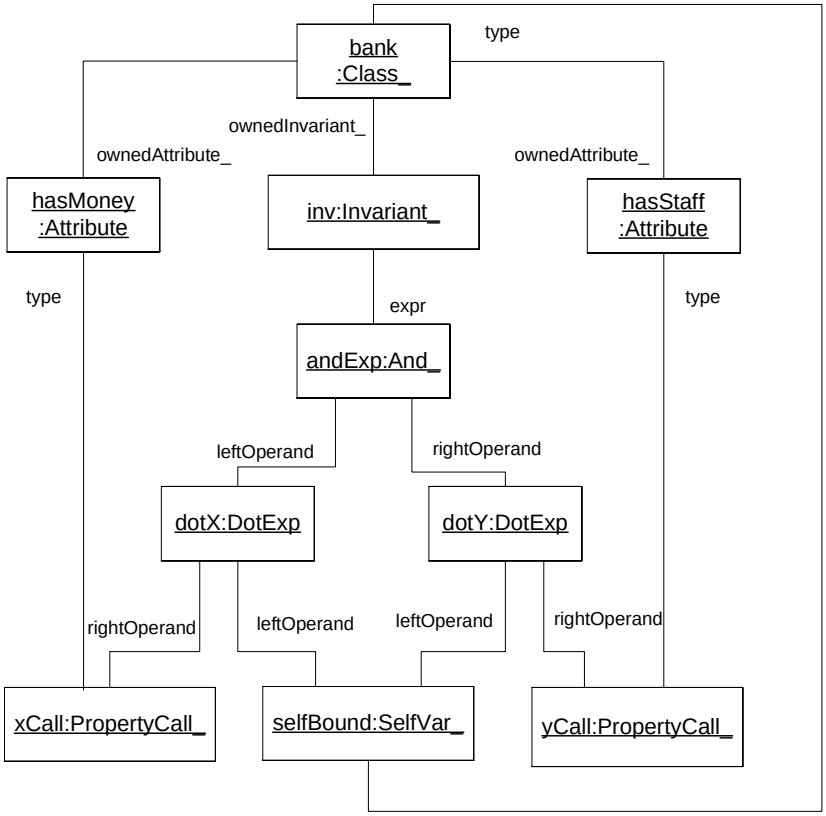

Fig. 8. Example (syntax) snapshot of a constraint

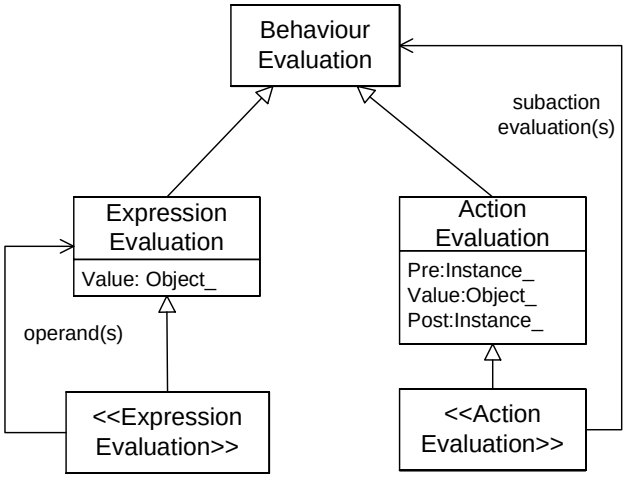

Fig. 9. Overview of a (semantic domain) architecture to support the interaction of expressions and actions

6. Related work

Other work has explored providing a definition for the meta-model of OCL. Most notable is the work presented in [1]. Although our resulting definition closely mirrors that work, the important characteristic of our approach is the use of templates to arrive at the definition. We have found that once the templates have been developed, then it was a relatively small step to defining OCL.

The work described in [11] is an early attempt at providing a meta-model for OCL. Unlike our work, this definition does not provide a separation between the syntax and the semantics of the definition. The well formedness rules are also described informally which compromises the precision of the definition.

7. Conclusion

In this paper we have introduced an approach to rigorously defining the OCL component of UML 2.0. The novelty of our approach lies in the use of templates to arrive at the definition. Our experience of using templates both within the context of the work presented here, and within the wider scope of defining the 2U submission for UML 2.0, suggests that they are formidable approach to developing complex languages from simple manageable units. Confidence in the approach is augmented by the use of MMT which supports the building of models. Further to our definition of OCL expressions for describing constraints, we have briefly outlined the interaction between computational OCL expressions and actions. We are currently in

the process of building an action language using our action definition. This will enable us to verify that our definition of actions meets the requirements and also to understand better the interaction of actions with expressions.

Acknowledgments

This research was generously funded in part by TATA consultancy services (India).

References

1. Boldsoft, Rational Software Corporation and IONA , *Response to the UML 2.0 OCL RFP (ad/2000-09-03)*, 2000.
2. Object Management Group , *OCL Request for proposals*, Available from http://cgi.omg.org/cgi-bin/doc?ad/00-09-03, 2000.
3. T. Clark, A. Evans and S. Kent , *Unambigious UML (2U) Revised Submission to UML 2 RFP*, Available from: http://www.2uworks.org, 2002.
4. T. Clark, A. Evans and S. Kent , *A programmers guide to MMT*, Available from: http://www.dcs.kcl.ac.uk/staff/tony/docs/ProgrammersGuideToMMT.pdf, 2002.
5. 2U Consortium , *http://www.2uworks.org*.
6. D. Harel and R. Bernhard , *Modeling Languages: Syntax, Semantics and All That Stuff*, The Weizmann Institute of Science, Rehovot, Israel, 2000.
7. E. Mikk, Y. Lakhench, C. Petersohn and M. Siegel , *On formal semantics of statecharts as supported by STATEMATE*, Springer-Verlag, 1997.
8. J. M. Spivey , *Z Notation*, Second ed., Prentice Hall, 1992.
9. E. Gamma, R. Helm, R. Johnson and J. Vlissides , *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
10. B. K. Appukuttan, T. Clark, A. Evans, G. Maskeri, P. Sammut, L. Tratt and J. S. Willans , *A pattern based approach to defining the dynamic infrastructure of UML 2.0*, Kings College, London, 2001.
11. M. Richters and M. Gogolla, A metamodel for OCL, in *Second International Conference on the Unified Modeling Language: UML'99*, R. B. France and B. Rumpe, Eds., Springer, 1999, Vol. LNCS 1723.