The Object Constraint Language for UML 2.0 – Overview and Assessment

Heinrich Hussmann
University of Munich
Munich, Germany
e-mail: Heinrich.Hussmann@informatik.uni-muenchen.de

Steffen Zschaler
Dresden University of Technology
Dresden, Germany
e-mail: Steffen.Zschaler@inf.tu-dresden.de

February 2, 2004

Abstract

In parallel to the release of UML 2.0, also a new version of the Object Constraint Language (OCL) has been published. OCL is a language for precise textual description of constraints which apply to the graphical UML models. The new OCL 2.0 standard goes far beyond the previous language, not so much in features but mainly in the approach chosen for laying much more precise and formal foundations for the language. This paper, authored by members of the OCL 2.0 team, gives an overview of the new aspects of OCL 2.0 and also provides a critical discussion of a few selected aspects of the language.

1 Introduction

The Unified Modeling Language (UML) [5] has the goal of providing a standard notation for all aspects of modeling in Software Engineering. Most of this modeling is done in a graphical way using various diagrams. However, since version 1.1 of the standard, UML contains a purely textual sublanguage which serves as a tool to express properties which are too complex to be expressed in a diagram adequately. This language, the Object Constraint Language (OCL) [13, 14], is targeted at two different areas of application. On one hand, it finds its main application in the definition of the UML standard (and a few other OMG standards), being applied on the meta-level and being used, e.g., for expressing well-formedness conditions for UML diagrams. On the other hand, OCL is also suitable as a lightweight replacement for formal specification languages (like Z, VDM, OBJ etc.). In this case it is applied on the level of application modeling, where it is used to express properties like invariants of the static class model, pre- and postconditions for operations and guards for state transitions. Although OCL has not yet gained much popularity for formal specification in software development, it has been recoginzed as one of the most interesting candidate languages for programming by contract, e.g., for platform-independent description of software components [2, 1].

In the past, OCL has been criticized frequently for the fact that this language, which aims at improving the formal preciseness of UML, does not have any formal basis for itself. Various suggestions for a more precise semantics of OCL have been produced over the years, for instance [12, 7, 6].

With the preparation of version 2.0 of the UML standard, there was a chance to react to the criticism and to improve the formal basis of the language. The OMG had put out separate calls for the improved OCL specification, independent of the UML core standardization process. This made it relatively easy to form a consortium of people (from industry and academia) which were specialized in the technical issues of OCL and in particular in the foundation issue. The authors of this contribution were part of the OCL 2.0 team and report here about the innovations in the language. Moreover, a critical discussion of a few selected aspects of the language is given. Therefore, the paper is structured into two main sections dealing with these two topics.

2 What's new in OCL 2.0?

In general, the transition to OCL 2.0 [4] has been defined in a way which hopefully does not disturb too much the slow process of adaptation of OCL. There are just a very small number of additional language constructs which are particularly useful for OCL as a contract specification language. Besides that, the main effort went into improving the formal basis of the language.

2.1 Metamodel

The concepts of OCL 2.0 as well as the relationships between these concepts have been expressed in the form of a MOF metamodel. Because UML is also formulated as a metamodel, this allows a much cleaner integration of OCL 2.0 into the UML. In addition, the presentation is much more formal and allows mapping to a semantic domain much more easily than previous versions of the OCL, which had no metamodel representation.

The OCL metamodel consists of two major parts: the types package and the expressions package. The types package defines the types which are recognized by OCL and relates them to core UML concepts. It may be worthwile to point out that every type is a super type of `OclVoid`, which implies that the undefined value `OclUndefined` is an element of every type in OCL. This includes the Boolean type which means that OCL 2.0 (as its predecessor versions) has a three-valued logic instead of the more common two-valued logic. OCL defines all operations and expressions to be *strict*, except where something else is explicitly specified. An expression is said to be *strict* if it evaluates to `OclUndefined` whenever one of its parameters is undefined.

The expressions package is for the most part the same as in older versions of OCL. The most important new expression types are OclMessage expressions (cf. 2.4) and tuple related expressions.

2.2 Query Language

With version 2.0, OCL has moved from a constraint language to a full query language for object-oriented models. The most prominent witness of this change is the introduction of the tuple type. This type allows it – similarly to the record/struct types in many programming languages – to combine multiple values of different types into one value. Thus they can be transported together through iterate-expressions which allows essentially the full range of query expressions to be formulated.

Another outcome of the move towards a query language is chapter 12 of the specification, which explicitly describes the association of OCL expressions to a UML model. In difference to previous versions of the language, the actual specification of the *language* OCL is given independently of the definition of the options for *usage*. In addition to the standard usage types invariant, pre- and post-condition, or guard, the specification also defines the use as initial or derived value expression, operation body expression, or definition expression, all of which use OCL as a generic query language.

2.3 Formal semantics

OCL 2.0 is the first version of OCL which has a defined, and standardized semantics. Although there still remain some questions and issues this is an important step towards a truly formal constraint language for the Unified Modelling Language. Tool vendors now have the necessary information to build sophisticated analysis and proof tools for UML/OCL specifications.

In fact, there are even two definitions of the semantics of OCL in the specification. The first is based on [8] and uses a metamodelling-based approach to the definition of the semantics. It provides a metamodel for the semantic domain and uses well-formedness rules to map instances of the abstract syntax metamodel onto instances of the semantic domain metamodel. The second definition of the semantics is based on Mark Richters' PhD thesis [11] which uses naïve set theory to formulate its semantic domain.

The specification claims that the two definitions of the semantics are equivalent. However, only the metamodelling based approach has normative character.

2.4 OclMessages

The OCL 2.0 specification introduces the concept of message expressions. These can be used to state that a certain message has been sent from a classifier during a certain period of time. A typical application of this concept is to specify that an operation call has been sent over a certain port to another component, the precise semantics of which is not known. The definitions of this concept are based on work in [9]. In the semantic domain, every snapshot is extended with an input and an output queue which hold events (or `OclMessageValues`) which have been sent to or from the corresponding object (represented by an `ObjectValue` in the semantic domain). The object constraint language provides constructs to query the contents of the output queue, so that constraints on the messages sent from a classifier can be specified.

The following two types of statements are provided:

`exp^^op (params)` The intuitive meaning of this statement is that it extracts the messages matching the pattern `op (params)` from the output queue of the object represented by `exp` and makes them available for further inspection as a collection. Each parameter can be either a complete expression or a term of the form `? : type` where the specification of the type is even optional. Messages in the output queue are matched first by the name of the operation or signal (`op`), then by number and type of the parameters (Including the `?` parameters. If no type has been specified for these parameters they can match any type.), and then current parameter values where full expressions have been given in the pattern.

`exp^op(params)` This is a convenience shortcut for `exp^^op (params) ->size() = 1`.

3 Problems in OCL 2.0

After giving an overview of the new properties of OCL 2.0, we want to describe some of the problems which still remain to be solved. We will briefly look at three areas of the OCL, namely:

- OclMessages,
- commonSuperType
- the representation of the concrete syntax

3.1 OclMessages

The concept of message expressions is still very new in the OCL. Consequently, there are still some issues which may be improved. We want to point out two more fundamental issues here, other minor issues have to be solved as well.

First, because OCL has no way to specify free variables, message expressions are restricted to two types of constraints. We can either specify that a certain signal has been sent or that a certain operation has been called *without constraining the parameter values in any way*, or we can specify that a signal has been sent / an operation has been called with explicitly stated parameter values. Of course the two variations can be mixed, but it is impossible to use more powerful constraints on the parameter values. For example, it is impossible to state that operation `op` has been called with a parameter which was less than 10. Of course, we could try to use explicit existential quantification, phrasing the above example as `Real.allInstances()->exists (r | op^(r) and r < 10)`, but this only works for types which actually support the `allInstances` operation (and therefore not for our example).

Secondly, it is often useful and important to be able to express constraints on the order of messages sent from a classifier. For example, in an application of the "Observer" pattern [3] to an implementation of a list, it is important to specify for the `remove` operation whether the corresponding event is sent out before or after the actual removal. Because OCL only allows to select messages with the same name in one expression and because we cannot compare the relative times of occurence for messages selected by different expressions, there is no way to specify such a constraint with the current OCL. In fact it is unclear, what happens if an operation is called more than once between pre- and post-condition time. (Unfortunately, the formal semantics of OCL 2.0 does not cover message expressions, so there is no clarification from there.) Essentially, there are two options:

- `exp^^op (params)` is restricted to have a maximum size of one. In this case we need to define which message is selected. If this is done in an indeterministic way, then constraints on the parameters are hard to provide. For example, what is the value of `let a:Integer = 7 in adder^add (a)` if two

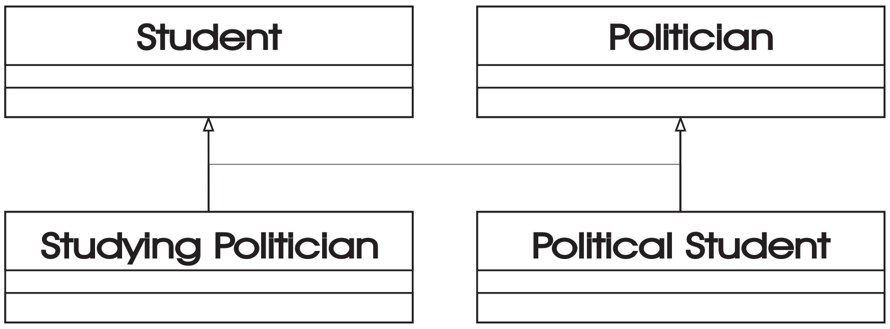

Figure 1: commonSuperType example

add-messages have been sent, one with a parameter of 7 and one with a parameter of 6? Is it undefined? Or is it false?

- `exp^^op (params)` contains a subsequence of the output queue which represents all the matching messages in the order in which they appear. In this case one could actually distinguish different messages, reason about their order, and could also handle the case where different invocations use different parameters.

3.2 commonSuperType

The OCL 2.0 specification defines the operation `commonSuperType` on `Classifier`. This operation is used in a process called type inference. Type inference is the attempt to identify the type of an OCL expression from the expression directly without any explicit type information being given. For example, the expression `Set{0, 1, 3.5}` can be inferred to be of type `Set(Real)` using the type inference rules from the specification.

However, consider the class diagram in Figure 1 and the following OCL expression:

```
let a: StudyingPolitician = ... in
let b: PoliticalStudent = ... in
Set{a, b}
```

What is its type? The OCL 2.0 specification states on page 52 that the element type of a collection literal expression (`Set{a, b}` is such an expression) is the common super type of all element expressions as determined by the `commonSuperType` operation. From the definition of this operation it follows that the type of the above expression cannot be determined precisely. Because both `Politician` and `Student` are common super types of `a` and `b`, `Set{a, b}` could have either one type of `Set(Politician)` or `Set(Student)`. The intuition of this definition of `commonSuperType` is clearly that the type of the expression should be determined depending on how the expression is *used*. However, this is beyond the scope of the well-formednes rules given in the specification.

3.3 Concrete Syntax Representation

Because the OCL 2.0 specification for the first time defines an abstract syntax in the form of a MOF metamodel, a mapping from the concrete syntax (e.g., the grammar for the textual represantation of OCL expressions which is given in the specification) to the abstract syntax needs to be provided. In order to facilitate this, the grammar provided has been extensively reworked and written down in a format which was invented specifically for this purpose. There are two problems with this approach: a) it is hard to impossible to understand how this grammar relates to the grammars of previous versions of OCL, and b) Ansgar Konermann showed in [10] that it is impossible to derive a working parser from this specification without a good deal of guesswork while resolving disambiguities in transforming the grammar into a LR/LALR form.

4 Conclusions

The revision of an accepted standard is always a difficult undertaking, and unfortunately such an effort usually is made without a clear beforehand evaluation of the user perceptions and the actual issues which need improvement. In the case of OCL 2.0, there was at least one very obvious requirement, which was the improvement of the formal basis. Significant progress has been made in this respect. However, one has to admit that the document describing the language has grown enormously in size. Fortunately, only a few people, for instance tool developers, actually have to read the main parts of the formal definition. Still, the attempt to thoroughly apply the UML and OCL metamodeling approach has led to a more complex syntax definition than probably was necessary. Regarding language features, the revision has closed a few obvious gaps. Key improvements are a proper model query language and a better support for the interaction of a component with its "used" interfaces (which is the main application of OCL message expressions). Nevertheless, as it has been shown above, several issues are still open. There will be need for extensive feedback from the scientific and industrial community to further stabilize this important sublanguage of the standard specification language UML.

References

[1] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison Wesley, 2001.

[2] Desmond D'Souza and A. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1998.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing. Addison Wesley, 1994.

[4] Object Management Group. UML 2.0 OCL specification. OMG document ptc/2003-10-14, October 2003.

[5] Object Management Group. UML resource page. http://www.omg.org/uml/, 2003.

[6] Rolf Hennicker, Heinrich Hussmann, and Michel Bidoit. On the precise meaning of OCL constraints. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, volume 2263 of *LNCS*, pages 70–85. Springer, 2002.

[7] Rolf Hennicker, Bernhard Reus, and Martin Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In *Proc. FASE 2001 – Fundamental Aspects of Software Engineering*, volume 2029 of *LNCS*, pages 285–300. Springer, 2001.

[8] Anneke Kleppe and Jos Warmer. Unification of static and dynamic semantics of UML: a study in redefining the semantics of the UML using the pUML OO meta modelling approach. available for donwload at: http://www.klasse.nl/english/uml/uml-semantics.html.

[9] Anneke Kleppe and Jos Warmer. Extending OCL to include actions. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.

[10] Ansgar Konermann. Entwurf und prototypische Implementation eines OCL 2.0-Parser. Masters thesis, Dresden University of Technology, 2004. In German.

[11] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Ph.d. thesis, Universität Bremen, 2002.

[12] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In *Proc. 17th Int. Conf. Conceptual Modeling (ER '98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.

[13] Jos Warmer and Anneke Kleppe. *The Object Constraint Language. Precise Modeling with UML*. Addison Wesley, 1999.

[14] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, 2003.

Curricula Vitae

Heinrich Hussmann holds a diploma degree in Computer Science from Munich University of Technology and a doctoral degree from University of Passau. He did research and teaching work at universities in Munich, Passau and Dresden. For several years, he was a systems engineer and team leader in the advanced development laboratory of the telecommunications division of Siemens. From 1997 to 2002 he was full professor for Computer Science at Dresden University of Technology, and since March 2003 he is full professor for Computer Science (Media Informatics) at the University of Munich (LMU). He participated in over 10 national and international projects in the area of software engineering and telecommunications, and is author of over 50 scientific publications, including three internationally published books

Steffen Zschaler obtained his Dipl.-Inf. from the Dresden University of Technology in 2002 with a thesis on the practical application of OCL in software projects. He is one of the co-authors of the OCL 2.0 specification. Currently he is working as a research assistant at the Dresden University of Technology in the field of non-functional properties of component-based software.