Experiences with the UML/OCL-Approach to Precise Software Modeling: A Report from Practice

Thomas Baar
Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, Germany
email: baar@ira.uka.de

Abstract

This paper is concerned with the practical usability of the Object Constraint Language (OCL). Pitfalls for untrained persons are uncovered, and strategies for avoiding them are discussed. These strategies are not restricted to OCL-specific problems but give insights how to handle formal specification, modeling, implementation, and verification issues within a single framework. The implementation of the identified strategies and their integration into a widely used commercial CASE tool is currently under way within the KeY project.

1 Introduction

Established formal techniques like VDM, Z, and B advocate a particular approach to software development. They start with an abstract specification which must be successively refined to more concrete ones. The logical foundations of these techniques are well elaborated, necessary proof obligations during refinement are generated automatically. Despite the recent success gained with formal methods, I see the following main drawbacks: usage of a purely mathematical specification language, the restrictive way of refinement, a lack of treatment of OO-concepts, and the high demands on the skills in formal methods of prospective users.

The application of the Unified Modeling Language (UML) to the design of software systems is becoming more and more popular. In the last few years, UML evolved into a widely accepted standard notation in the area of industrial software development.

In recent years a lot of effort was put into bringing the UML and formal methods together. The incorporation of formal methods into the UML approach is tried at several levels. For specification purposes UML was extended with OCL to make UML models more precise. For verification purposes a formal semantics of UML and OCL is currently under development, mainly pursued by the Precise UML group (pUML) [3].

Yet, there is no UML tool able to handle specification and verification issues in such a complete and elaborate way as existing commercial tools for B, VDM, and Z. This paper describes implicitely a (not necessary complete) catalog of requirements for an appropriate UML tool and a conceivable realization. The requirements are the result of observations in practice and are formulated as improvement strategies targeting frequent mistakes.

Section 2 describes the mistakes practitioners made using UML and OCL as design and specification notation and focuses on the reasons for occurring mistakes. Some reasons have a very unspecific nature (e.g. time pressure during development, insufficient structure of design documents) but others are highly specific (e.g. semantics of OCL constructs are not understood). The analysis of the reasons for mistakes allows in Section 3 the definition of improvement strategies. An improvement strategy should help to avoid or to uncover mistakes.

Mistakes can be avoided if, for instance, (1) suitable specification methods are recommended in particular situations, (2) a catalog of standard specification problems and solutions is available, and (3) the intuitive semantics of some OCL constructs is made clearer.

Mistakes can be uncovered by, for instance, (1) passing the specification through a syntax and validation check, (2) animation, (3) testing of the resulting program code with established techniques, and, finally, (4) verification of program code against specification.

The identified improvement strategies can be implemented as a part of a CASE tool. Such an implementation is currently under development as part of the KeY project [4] at the University of Karlsruhe (Germany) and the Chalmers University of Technology in Göteborg (Sweden). The resulting KeY tool is an extension of the commercial CASE tool TogetherJ [11].

2 Observations in Practice

After a brief description of the concepts of OCL I report about mistakes software developers made using UML and OCL. I give a classification of mistakes and identify some reasons for why they were made.

2.1 Concepts of OCL

The basic idea for incorporating formal details into UML models is extending the UML instead of keeping the formal parts separate. For that purpose the OCL was developed. OCL allows (among other things) the specification of pre-/post-conditions and invariants. It provides a variety of specification techniques. The main roots of OCL are:

1. **Set theory** : The collection of all instances of a class is regarded as a set. Also the result of the navigation from a ClassA to a ClassB via an association is a set of instances of the ClassB.

   Set theoretic concepts like cardinality (`size`), set comprehension (`select`), set projection (`collect`), and set algebra operators (`union`, `intersection`, etc.) are supported.

2. **Predicate logic** - OCL provides the type `Boolean` together with the usual logical connectives (`and`, `or`, `not`, `implies`). Furthermore quantifiers (`forAll`, `exists`) are available but restricted to a given set. This corresponds to a concept of dynamic sorts in predicate logic.

3. **Operational semantics** - Computational aspects can be expressed using the `iterate` construct. It offers possibilities which go beyond the ones provided by set theory and predicate logic. A typical application is to sum up all elements of a given set of integers.

A more detailed description of OCL can be found in [12, 9]. OCL is a very young language and not all its concepts are incorporated into a single logical framework yet. Furthermore, the intuitive semantics provided by Warmer/Kleppe [12] contains some contradictions [10].

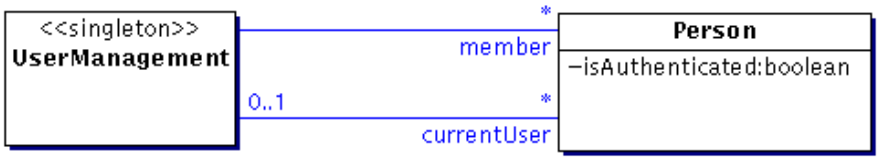
2.2 A Catalog of Phenomena and Reasons

An inconsistent or poor design always contains a lot of contradictions, inelegant structures, redundant information, etc. I prefer for these symptoms the term *phenomenon*. The first step towards a better design technique is to ask why a phenomenon arised.

Classified according to the identified reason lists of phenomena are listed below. I illustrate some of the most interesting phenomena with an example.

Lack of Concentration

This reason can cause the phenomena *typing mistake* and *syntactical error*, both in specifications and implementation code. Examples are valid but misspelled identifiers, invalid identifiers (e.g. calling a method which is not public or does not exist), or incomplete OCL expressions like in the following example:



For this design, the next incomplete constraint was observed:

```
context UserManagement
    -- each current user is registered (member)
    members->includesAll(currentUsers) and
    -- each current user is authenticated
```

Here, the OCL formulation of the second part

```
currentUsers->forAll(x | x.isAuthenticated)
```

is missing.

Excessive Complexity

If a design becomes too complex inconsistencies are unavoidable. Designers may tend to draw real complex designs just by adding more details over time. Especially, interconnections between classes can lose their structure easily. But – and this is the interesting point – designers do not overwhelm

OCL constraints in the same way. OCL constraints keep their simple structure. The main problem with OCL constraints is that they may get redundant or contradictonary when the underlying UML model is changed.

Misunderstanding of OCL Semantics

This reason causes rather obvious mistakes like

```
aBag->forAll(x, y | x <> y)
```

Such a constraint does not make sense, because in almost all circumstances (except `aBag` is the empty set) it will be evaluated to `false`[1]. The nested `forAll` operator takes every possible evaluation of `x` and `y` into account, also the evaluation to a same element, lets call `elem1`. Because of this the `forAll` expression also claims `elem1 <> elem1` what is obviously `false`.

The software developers often misunderstand the semantics of the nested `forAll` construct and assume that `x` and `y` cannot be the same. The intended constraint, however, can be expressed in OCL too:

```
aBag -> forAll(x | aBag->count(x) = 1)
```

The misunderstanding of the nested `forAll` construct seems to be a very common mistake. Even the authors of the official semantical documents for OCL made the same mistake in the definition of the `isUnique` predicate (see [9] page 7.37).

Weakening of Logical Expressions

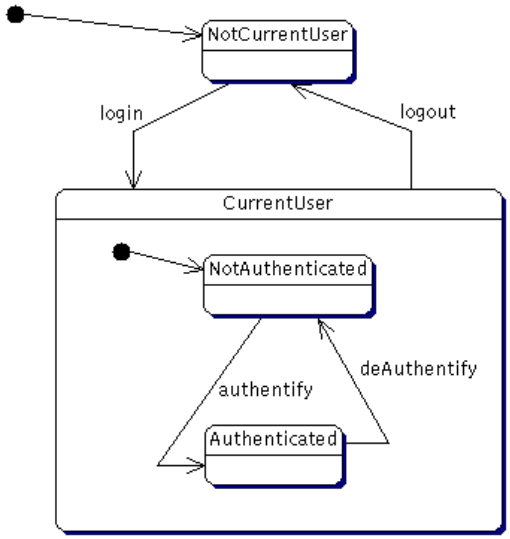
Some software developers may have difficulties to decide which of two logical expressions is the weaker one. This reason causes errors in specifications of invariants and pre-/post-conditions. The correct weakening of expressions is crucial for the substitution principle within an inheritance structure. The substitution principle is the core upon all polymorphic concepts of object orientation are based. Among the programming languages only Eiffel [8] supports the substitution principle in form of *design by contract* [7]. Despite its weak acceptance at implementation level the substitution principle becomes crucial at design level.

Unclear issues in OCL semantics

OCL contains some rather unintuitive concepts and restrictions like prohibition of nested sets, and the indeterministic iterate-construct. If designers do not understand these concepts they use again natural language for specification or a private list of ad hoc predicates and operators. In the consequence, the specification contains a lot of ambiguities. An example where OCL is not expressive enough, is a post-condition of a method which creates a new object (e.g. 2-dimensional array). The `new`-construct can be simulated in OCL but is not part of the standard. Furthermore, the new array cannot be described intuitively as a collection of lists, because this would result in a nested set, which is not allowed.

Insufficient Semantical Interconnections

The semantics of OCL and other specification techniques like StateTransition or Activity diagrams are not defined in a common formal way. Once designers learnt the concepts of OCL they tended to use OCL also for purposes where a StateTransition or Activity diagram would be more suitable. Consider the following example:



[1] Also evaluation to `undefined` is possible namely if `aBag` is `undefined`. Due to simplicity I omit this possibility here.

Instead of drawing such a StateTransition diagram the designers specified pre-/post-conditions of occurring operations, e.g. for login():

```
context Person::login()
  pre:  not UserManagement.currentUsers->includes(self)

  post: UserManagement.currentUsers->includes(self)
```

The specification of the methods `authentify` and `deAuthentify` requires furthermore the encoding of the states `(Not)Authenticated` with a special flag.

The OCL technique is very implementation oriented and error prone. Nevertheless the developers prefer it because it is not clear to them how OCL pre-/post-conditions could be generated from a StateTransition or Activity diagram.

3 Improvement Strategies

In the previous section I identified common types of phenomena and possible reasons for them. This analysis allows the formulation of improvement strategies aiming at getting rid of the negative phenomena.

3.1 A coarse Classification

The identified reasons can be grouped into three categories: human (in)capabilities, personal, and external ones. For each group there exist trivial strategies to avoid them or to minimize negative consequences.

For the first two reasons identified in Section 2.2, the human nature is responsible. The development of avoiding strategies is a research topic in psychology and has a strong influence on management techniques. The discussion of psychological issues is out of the scope of this paper. However, some strategies to uncover phenomena (not to avoid the reasons) are given in Section 3.2.

For the next two reasons, the personal (in)abilities of a particular software developer are responsible. An obvious improvement strategy for that is having more training in mathematics, logics, etc.

The last two reasons cannot be personalised to the application developer. They are external and more research have to be done to avoid them.

3.2 A Catalog of Strategies

In the following I discuss improvement strategies which can somehow implemented as part of a CASE tool. Each improvement strategy presented here is an answer to one or more phenomena listed in Section 2.2. Some improvement strategies aim at avoiding situations for the developer which can be a reason for a phenomenon. These strategies avoid errors. Other strategies aim to find phenomena. These strategies uncover errors.

Syntax Check

The incorporation of a syntax checker (e.g. an OCL parser) helps to find typing mistakes and syntactical errors. This is considered good practice in most areas of specification and programming languages but not integrated yet in the present generation of UML CASE tools.

Online catalog of specification idioms

For frequently recurring specification tasks (e.g. an element of a bag does not occur more than once, for all instances of a class the value of attribute `id` is unique, a datastructure is not cyclic, etc.) both the informal description of the constraint and its formalization in OCL should be pre-defined in the CASE tool. Then, the user can search for idioms he or she needs in a particular situation and the CASE tool automatically includes the formalization of the selected idiom into the specification, possibly with adaptions for the current situation.

Proof of Consistency

Consistency should be proven at UML level formally for invariants and pre-/post-conditions. This check uncovers typing mistakes and logical errors. A consistent specification is a necessary prerequisite for every verification activity. Starting with an inconsistent specification, arbitrary properties of the developed design or implementation can be concluded, even formally. This activity is usually called *horizontal verification*.

Proof of Substitution Principle

Checking the proof obligations arising form the design by contract approach reveals typing mistakes and logical errors. The main advantage, however, is the pressure on designers to make pre-/post-conditions explicit, which results in a clearer design. This activity is again part of the horizontal verification and is only concerned with modeling at the UML level, not with the implementation code.

Test Code Generation

Another idea of design by contract is to check specified pre-/post-conditions and invariants during runtime of the program. The language Eiffel provides this technique as a feature of the language. For other languages there exist tools for automatic generation of additional test code (e.g. iContract [5] for Java).

Verification

Using a program logic the implementation can be verified to have the properties expressed in the design. The motivation for this technique is the same as for the test code generation. The difference is the effort and the quality of the result: verified instead tested code. This activity is usually referred to as *vertical verification* because both the UML model and the implementation code are involved.

The strategies proposed so far can be implemented in a certain way within a CASE tool (see Section 4). The next two strategies address topics of research. They are less practical at the first glance, however, successes gained here will influence the next generation of CASE tools.

Uniform Semantics for all types of UML diagrams

This would make it easier to write animations and other tests which take all diagram types into account. As a consequence, inconsistencies among diagrams of different type could be detected.

Improved Intuitive Semantics of OCL

An improved informal semantics for OCL would have a great impact on usability. The already mentioned restrictions of some concepts have to be overcome. Other concepts mainly to specify behavior have to be added.

4 The KeY Approach

Despite its potential to produce trustworthy software the application of formal methods in industrial software development is still very rare. Within the KeY project we analyzed the reasons for that situation. As a result, we are currently designing and implementing a CASE tool unifying the UML approach to software development with formal methods. The user of the KeY tool will be able to develop correct implementation code, that is the tool supports specification/modeling of a system (using the UML) as well as implementation in the target language (e.g. Java), and it manages all verification issues. The user does neither have to learn a new specification language nor to know the logical details of the necessary verification process.

The interface of the KeY tool takes two very important facts into account: (1) Developers do not want to leave their environment of software development (2) To gain acceptance, the usage of formal methods is encouraged but not enforced.

To meet these requirements, a widely accepted commercial CASE tool is the heart of the KeY system. Due to its extensibility the CASE tool TogetherJ was chosen for that purpose. In Figure 1 you see the KeY tool in a situation where an incorporated OCL parser is called. The user interface of the underlying TogetherJ tool is almost kept unchanged. Only the menu group *KeyExtension* is added. All TogetherJ features can still be used, KeY only adds some few but powerful functionalities.

The goal of the KeY project is to have a tool which comprises all the improvement strategies listed in Section 3.2. The improvement strategy *Syntax Check* can be implemented easily using existing OCL parsers. In order to implement the strategy *Idioms*, basically only an interface for searching and inserting of text must be provided. The catalog of known idioms records experiences gained in previous projects and can be enlarged continously by the user.

The implementation and integration of strategies concerned with horizontal and vertical verification into a CASE tool requires profound theoretical foundations. For horizontal verification we need a formal logical model of what the UML model and the OCL constraints mean. Existing approaches like [6], [2] propose temporal logic as a framework. They are promising starting points but do not meet all practical needs to handle UML models. For vertical verification a program logic is needed which covers the semantics of the used implementation language formally. This logic must correspond with the logic used to express the semantics of UML models, because in vertical verification both implementation code (program logic) and its specification with UML statements (logical model of UML) are involved. Therefore we are currently working on a single framework based on dynamic logic which is able to cover implementation and specification issues [1]. For further information about KeY check the project website [4].

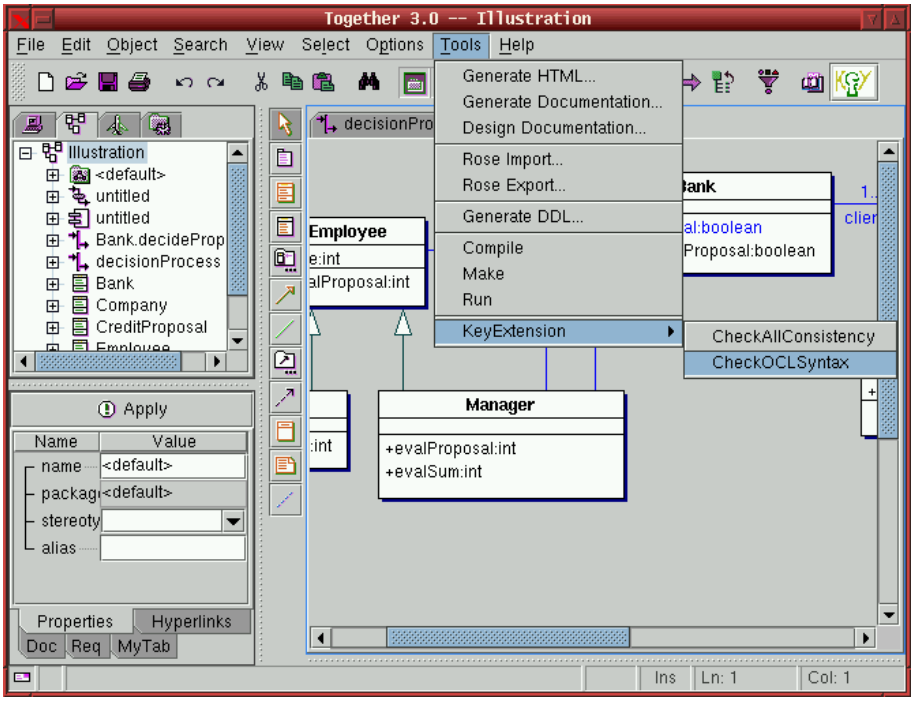


Figure 1: TogetherJ with the KeY extension

5 Conclusions

The primary motivation for this work was to gain experience with the use of OCL as a specification language in the practical software development process.

A first aspect is the applicability of OCL by software developers with a weak background in mathematics and formal methods. The insights I gained strongly recommend the usage of OCL. Developers do not hesitate to use it. The similarities of OCL notation and programming languages have a very positive psychological effect. OCL is much closer to the way developers think than more mathematical notations.

A second aspect is the expressive power of OCL. This is still rather restricted and semantical ambiguities of already integrated concepts cause a lot of difficulties. But these drawbacks can be overcome by an improved definition of semantics.

Furthermore, this paper addresses problems arising from writing and exploiting OCL constraints. Frequent mistakes have been listed together with their reasons. Possible strategies to avoid and uncover them have been shown.

The lessons learnt about OCL usage are taken into account in designing the KeY tool, a CASE tool of the next generation unifying approaches of UML and formal methods. So, we ensure that the KeY tool is oriented towards by the needs of practitioners and overcomes a main drawback of existing formal methods tools.

References

[1] Bernhard Beckert. A dynamic logic for java card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, 2000. To appear.

[2] Juan Bicarregui, Kevin Lano, and Tom Maibaum. Formalising object-oriented models in the object calculus. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, pages 45–51. Technische Universität München, TUM-I9725, 1997.

[3] Andy Evans and Stuart Kent. Core meta-modelling semantics of UML: The pUML approach. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[4] KeY. *Project KeY - Integrated Deductive Software Design*. University of Karlsruhe and Chalmers University Göteborg. Information available at: `http://i12www.ira.uka.de/~projekt/index.html`.

[5] Reto Kramer. iContract—the Java Designs by Contract tool. In *Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara/CA, USA*. IEEE Press, Los Alamitos, 1998.

[6] Kevin Lano and Juan Bicarregui. Formalising the UML in structured temporal theories. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 105–121. Technische Universität München, TUM-I9813, 1998.

[7] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[8] Bertrand Meyer. *Eiffel - The Language*. Prentice-Hall, Englewood Cliffs, 1992.

[9] Rational Software Corp. et al. *Unified Modelling Language Semantics, version 1.3*, June 1999. Available at: `www.rational.com/uml/index.jtmpl`.

[10] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449–464. Springer, Berlin, LNCS, 1998.

[11] TogetherSoft LLC. *TogetherJ tool*. Information available at: `http://www.togethersoft.com`.

[12] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.