Experiences with the UML/OCL-Approach in Practice and Strategies to Overcome Deficiencies

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

Abstract. This paper is concerned with the practical usability of the Object Constraint Language (OCL). Pitfalls for untrained software developers are uncovered, and strategies for avoiding them are discussed. These strategies are not restricted to OCL-specific problems but give insights into 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 [5], (Object)-Z [13, 12], and B [1] advocate a particular approach to software development. They start with an abstract specification which is 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 of formal methods, the author sees the following main drawbacks: usage of a purely mathematical specification language, the restrictive way of refinement, the mostly inadequate 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 getting more and more popular. In the last few years, UML evolved into a widely accepted standard notation in the area of industrial software development.

Recently, a lot of effort has been put into bringing the UML and formal methods together. 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) [4].

However, the potential of UML for precise modeling is not widely used yet. This paper illuminates some obstacles for practitioners when using UML and especially OCL. The identified issues are the essence of observations the author made during a joint project with a software manufacturing company. The motivation for that collaboration between industry and academic research was twofold: 1) Software developers from industry become acquainted with precise modeling techniques and the usage of OCL. 2) Academic researchers get experiences about problems in practice and can draw conclusions how to build better CASE tools.

This paper is organized as follows: Section 2 gives a brief introduction to OCL. Section 3 describes the mistakes practitioners made using UML and OCL as design and specification notation and focuses on the reasons for these mistakes. The analysis of the reasons allows the definition of improvement strategies in Section 4. 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 check, (2) animation, (3) testing of the resulting program code with established techniques, and, finally, (4) verification of program code against its specification.

Some of the identified improvement strategies can be integrated into a CASE tool as described in Section 5. Such an integration is currently under development as part of the KeY project [6] at the University of Karlsruhe (Germany) and the Chalmers University of Technology in Gothenburg (Sweden). The resulting KeY tool is an extension of the commercial CASE tool TogetherJ [14].

2 Concepts of the 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. The 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 class A to a class B via an association is a set of instances of B.

   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` with the usual logical operators (`and`, `or`, `not`, `implies`). Furthermore, quantifiers (`forAll`, `exists`) are available but are 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 going 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 [15, 10]. OCL is a very young language and not all its concepts are incorporated into a single logical framework yet. Furthermore, the informal semantics provided by Warmer/Kleppe [15] contains some contradictions [11].

3 Deficiencies observed in a real life Project

This section reports on some mistakes software developers made using UML and OCL. All mentioned mistakes were observed during a 6-month project in which two software developers from a software manufacturing company, one student, and the author were involved. The project's goal was the development of the high-level design of a new product (an email program). In this project only formal specification techniques were used for the sake of clarity. However, OCL constraints were only annotated as "formalized comments" because the employed UML CASE tool does not support OCL properly.

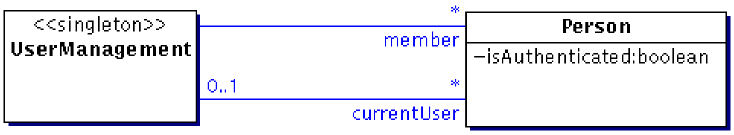
3.1 A Catalog of Phenomena and Reasons

An inconsistent or poor design always contains a lot of mistakes, contradictions, inelegant structures, redundant information, etc. In this paper the term *phenomenon* is preferred for these symptoms. The first step towards a better design technique is to ask why a phenomenon has arisen.

The observed phenomena are listed below classified according to the identified reason. The most interesting phenomena are illustrated with an example.

Reason 1: 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 that is not public or does not exist), or incomplete OCL expressions like in the following example:



For this design, the following 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.

Reason 2: Excessive Complexity

If a design becomes too complex the phenomena *inconsistencies* and *redundant information* are almost unavoidable. Designers tend to draw rather complex designs by adding more details over time. A typical effect is that interconnections between classes lose their structure. More important, however, 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.

Reason 3: Misunderstanding of OCL Semantics

This reason causes the phenomenon *semantical mistakes*. An example is:

```
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, say `elem1`. Because of this the `forAll` expression also claims `elem1 <> elem1` which 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 documents defining the semantics of OCL made the same mistake in the definition of the `isUnique` predicate (see [10], page 7.37).

Reason 4: Difficulties with Logical Weakening

Some software developers have difficulties to decide which of two logical expressions is the weaker one. This causes the phenomenon *incompatible specification of invariants and pre-/post-conditions within inheritance structure*. As a result, the substitution principle, upon which all polymorphic concepts of object orientation are based, is violated. Among the

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

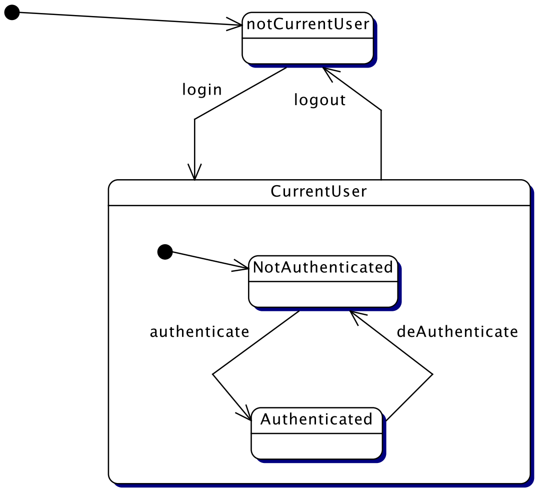
well-established programming languages only Eiffel [9] supports the substitution principle in form of *Design by Contract* [8]. Despite its weak acceptance at the implementation level the substitution principle becomes crucial at the design level.

Reason 5: Unclear Issues in OCL Semantics

OCL contains some rather unintuitive concepts and restrictions like prohibition of nested sets, and the non-deterministic iterate-construct. If designers do not understand these concepts they fall back to the *usage of natural language* for specification purposes. As a result, the specification contains a lot of ambiguities.

Reason 6: Insufficient Semantical Interconnections

The semantics of OCL and other specification techniques like StateTransition or Activity diagrams are not formally defined in a common way. This causes the phenomenon *exploitation of only one specification technique*. Once designers had learnt the concepts of OCL they tended to use OCL also for purposes where a StateTransition or Activity diagram is more suitable. Consider the following example:



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 `authenticate` and `deAuthenticate` with OCL requires furthermore the encoding of the states `(Not)Authenticated` with a special flag.

In such cases, 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.

4 Improvement Strategies

In the previous section common types of phenomena and their reasons were identified. This analysis allows the formulation of improvement strategies aiming at getting rid of the negative phenomena.

Each improvement strategy presented here contains in brackets one or more of the reasons listed in Section 3.1, indicating that the strategy aims to avoid or uncover phenomena caused by the referenced reasons.

Syntax Check and Static Analysis [1,3]

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. A static analysis can detect frequent mistakes using non-deductive methods.

Online Catalog of Specification Idioms [3]

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 could 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 adaptations to the current situation.

Proof of Consistency [2,4]

Consistency should be formally proven at the UML level 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 [4]

Checking the proof obligations arising from 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.

Vertical Verification [2]

Vertical verification is the bridge from a UML design to its implementation. Vertical verification normally uses both implementation code and specifications of (imported) operations. Therefore, it often reveals insufficiencies of the specification and enforces a refactoring to a clearer specification.

The strategies proposed so far can be implemented within a CASE tool (see Section 5). The following 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 [6]

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

Improved Intuitive Semantics of OCL [5]

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

5 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 goal of the KeY project is to have a tool which comprises all the improvement strategies listed in Section 4. 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 neither has to learn a new specification language nor to know the logical details of the necessary verification process.

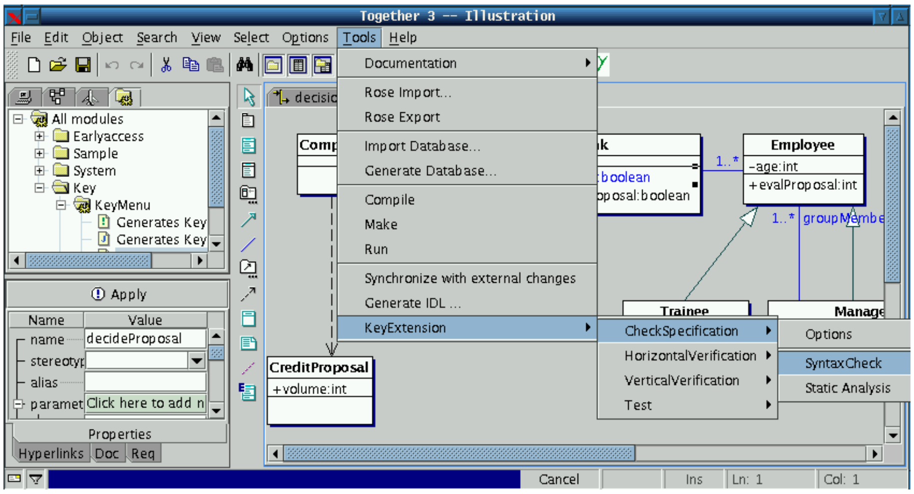


Fig 1. TogetherJ with the KeY extension

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

To meet these requirements, a widely used commercial and extensible CASE tool (TogetherJ) was chosen as heart of the KeY system. Figure 1 shows the KeY tool in a situation where an incorporated OCL parser is called. The user interface of the underlying TogetherJ tool is almost unchanged. Only the menu group *KeyExtension* and its submenus are added. All TogetherJ features can still be used, KeY only adds some powerful functionalities.

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 diagrams and the OCL constraints mean. Existing approaches, such as [7], [3], propose temporal logic as a framework. They are promising starting points but do not meet all practical needs to handle UML models. Vertical verification requires a program logic that covers the semantics of the used implementation language formally. This logic must correspond to the logic used to express the semantics of UML models, because in vertical verification both the 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 [2]. For further information about KeY check the project website [6].

6 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.

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

Another aspect is the accurateness of OCL's semantics. The mentioned insufficiencies and semantical ambiguities cause a lot of difficulties. However, 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. That way 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] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, August 1996.

[2] Bernhard Beckert. A dynamic logic for java card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, pages 111–119, 2000.

[3] 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.

[4] 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.

[5] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.

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

[7] 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.

[8] Bertrand Meyer. Applying “Design by Contract”. *IEEE Computer*, 25(10):40–51, October 1992.

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

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

[11] 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.

[12] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[13] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[14] TogetherSoft LLC. *TogetherJ tool*. See: `http://www.togethersoft.com`.

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