Applicability of the Object Constraint Language (OCL) in Commercial Software Development for Vehicle Routing and Scheduling Software

Peter Wendorff

ASSET GmbH
Am Flasdieck 5
D-46147 Oberhausen
p.wendorff@t-online.de

Abstract: Models are important artefacts that support human understanding and communication. Often software development involves specialists from a variety of fields, e.g. mathematicians, engineers, economists, software designers, and programmers. This may result in different models, and communication accross discipline boundaries requires to establish links between these models. In this paper we investigate this issue in the case of software development for vehicle routing and scheduling software. We concentrate on three artefacts: a mathematical specification of requirements, an object-oriented software design, and program code written in Java. Obviously these artefacts do not exist in isolation during software development. We demonstrate how the Object Constraint Language can be used to establish links between these three artefacts.

1 Introduction

Our company is presently involved in the maintenance of a successful commercial vehicle routing and scheduling (VRS) software package that is part of a comprehensive turn-key solution for trucks. This kind of software deals with the real-world version of the famous NP-hard "Traveling Salesman Problem (TSP)".

Whereas the TSP can easily be modelled by a few mathematical formulas, its real-world counterpart involves a number of additional factors, e.g. capacity constraints, time windows, dynamic planning, randomness, etc. Naturally, software that takes these aspects into account is very complex to specify, develop, and maintain.

VRS problems are at the core of operations research and management science. Software for this type of problem gets ever more important with the emergence and broad availability of new technology, e.g. geographic information systems (GIS) and larger fleets of vehicles. The more complex problems become, and the more complex information technology gets involved, the more human dispatchers must be supported by

powerful information systems to route their fleets economically. Therefore the market for these systems is burgeoning.

The typical model of vehicle routing and scheduling problems is mathematically orientated, consisting of an objective function that is to be optimised subject to a number of constraints. The observation that motivates this paper is that requirements for VRS software rely on constraints - and that the Unified Modeling Language (UML) incorporates a formal language called "Object Constraint Language (OCL)" (cf. [OMG00]). At first glance it seems that OCL and VRS might be an ideal match. The question that will be addressed in this paper is whether OCL can help to bridge the gap between the mathematical model, a software design model, and an eventual implementation during commercial VRS software development.

This paper describes the preliminary results of ongoing work that we have started in April 2000 to assess the applicability of OCL to VRS software development. In the course of that work an exploratory prototype has been implemented in Java to address the transition from OCL constraints to executable code.

In section 2 some basic ideas behind OCL will be stated. In section 3 some examples from our work will be used to demonstrate the integration of mathematical constraints in an object-oriented software design model through the use of OCL. In section 4 we will address issues that arise when OCL constraints are implemented using an imperative programming language. In section 5 we will state some commonalities and differences of OCL and the formal specification language Z to clarify the intentions behind OCL. In section 6 we will report about a free OCL type checker.

2 The Object Constraint Language (OCL)

In this section we will only hint at the ideas behind the Object Constraint Language (OCL), in particular as there is enough published material available on the subject, e.g. [OMG00], [Pr00], [WK99a], [WK99b].

OCL is based on the proven and popular concepts of precondition, postcondition, and invariant. We will follow the terminology introduced by B. Meyer in [Me97] and use the term "assertion" as broader term for the individual terms "precondition", "post-condition", and "invariant". Preconditions and postconditions date back to work of A. Hoare and E. Dijkstra in the 60s and 70s. Assertions are the informing idea of the "Design by Contract" paradigm of software development established by the work of B. Meyer, who has also designed a programming language called "Eiffel" that is centred around the implementation of assertions.

Usually an assertion expresses a property of a system that is a necessary condition for its integrity. If the integrity of the system is violated it cannot be assumed that the system will behave according to its specification. Assertions are usually based on properties of the system and first order predicate logic. OCL provides three essential things (cf. [Pr00, p. 210]):

- a way to adorn the graphical notation of UML with textual OCL assertions
- a syntax to access properties of the entities in an UML model
- a language to form predicates from these properties

Some other notable features of OCL are:

- OCL is a declarative language, and therefore it does not have an explicit flow of control. If OCL assertions must be implemented in an imperative language, then as a result problems may arise, because an explicit flow of control is a fundamental property of imperative languages (cf. [WK99a, pp. 84]).
- OCL does not specify what happens if an assertion is violated. Anyway, because it does not have side-effects any form of failure resumption model is precluded [WK99a, pp. 85]).
- OCL provides some primitive data types, namely Boolean, Integer, Real, and String.
- OCL provides collection types, namely Set, Bag, and Sequence, and these types come with a variety of powerful operations.
- OCL provides versatile iterators for the collection types.

3 Modelling of Vehicle Routing and Scheduling Problems

The classic representation of combinatorial optimisation problems is that of an objective function that is to be optimised subject to certain constraints that come in the shape of equations and inequalities. The objective function and the constraints involve a common set of variables, e.g. binary flow variables, time variables, and load variables (cf. [DDS91]). We will denote this approach and notation toward modelling for VRS software development a "mathematical model" in this paper.

Mathematical models are ideally suited to linear programming methods but less ideal for most other algorithms. VRS problems are NP hard, and even finding a feasible solution may already be an NP complete problem under certain conditions [So87, p. 255]. Therefore approximation algorithms are the only way to solve practical size VRS problems.

Before the emergence of object-oriented methods the typical implementation languages were stricly imperative in nature. For example the VRS software we are currently working on had been developed in Fortran 15 years ago, and it was reengineered and implemented in C 10 years ago.

In the absence of powerful implementation languages the usual software design has been based on the mathematical model, pseudocode, and simple data structures like matrices and vectors. There was little incentive for more sophisticated modelling techniques, because the programming languages did not provide matching language constructs. Clearly, the primitive implementation languages and the basic mathematical models have supplemented each other well in the past.

Our customer is considering to shift to C++ as object-oriented implementation language for the VRS software in the long term. With the shift to an object-oriented implementation language, the question is, whether the old and proven modelling techniques are still appropriate in conjunction with C++ as an implementation language. During our work we have found, that a mathematical model adorned with some pseudo code does

not make good use of the powerful abstraction mechanisms offered by an object-oriented language like C++. There is an "impedance mismatch" between modelling technique and implementation technique.

Our customer uses a sophisticated mathematical model for VRS software development. This model has been developed by a degree-educated mathematician and is clearly the most convenient and efficient means of expression for a person with such a background.

Many programmers are not degree-educated at all, and only a minority of programmers has a strong mathematical background. Therefore software development clearly necessitates a different kind of model to describe the relevant software artefacts and their relationships. Such a model is usually denoted a "software design model" in the industry, and we adopt this terminology in the following. These days the Unified Modeling Language is widely accepted as graphical notation for object-oriented software design models.

Accordingly, there are two separate models that we have to cope with during software development. On the one hand there is the mathematical model, on the other hand there is the software design model. Therefore there are two fundamental options: First, one could keep the two models separate and make them express different aspects of the VRS software. Second, one could try to integrate the two models.

The information conveyed by the mathematical model mainly comes in the shape of constraints on the variables. A part of the Unified Modeling Language is the Object Constraint Language (OCL). The objective of OCL is to provide a means to integrate quantitative constraints into an object-oriented software design model [WK99a]. Therefore it is a reasonable idea to use OCL to integrate constraints expressed by the mathematical model into an object-oriented software design model. This establishes links between the two models that can be used as basis for efficient requirements tracing [Ja98] during the whole software life cycle. In the following we demonstrate how this can be achieved.

A typical set of constraints found in VRS problems is given in [DDS91, (6) - (22)]. These constraints must be met by any feasible solution. From an operational point of view the constraints are postconditions of any algorithm that solves the modelled VRS problem. Some typical examples of such postconditions are:

$$\sum_{v \in V} \sum_{j \in N} X_{ij}^{v} = 1, \quad \forall\, i \in P^{+} \qquad \text{[DDS91, (6)]}$$

$$X_{ij}^{v} = 1 \qquad T_i + s_i + t_{ij} \le T_j\,, \forall\, i, j \in P^{-} \cup P^{+}, \forall\, v \in V \qquad \text{[DDS91, (12)]}$$

$$Y_0 = 0 \ \wedge\ d_i \le Y_i \le D, \forall\, i \in P^{+} \qquad \text{[DDS91, (21)]}$$

The constraints [DDS91, (6) - (22)] deal with a special variant of VRS problem, the so called "Pickup and Delivery Problem with Time Windows (PDPTW)". In this problem the network is modelled by a set N of nodes in an associated network graph. One node from N represents a depot, the set P^{+} represents the pickup nodes, and the set P^{-} represents the delivery nodes. The fleet V of vehicles must be routed through the net-

work. Each vehicle departs empty from the depot, visits a number of pickup nodes and delivery nodes and eventually returns to the depot empty again.

Constraint [DDS91, (6)] requires that any node from the set P^+ of pickup nodes must be served by exactly one vehicle v from the set V of all vehicles. In this expression X_{ij}^v denotes a binary decision variable, with $X_{ij}^v = 1$ if vehicle v travels from node i to node j, and $X_{ij}^v = 0$ otherwise.

Constraint [DDS91, (12)] expresses time constraints. In this the variable T_i denotes the time when vehicle v arrives at node i. The service time for pickup or delivery is s_i. After that the vehicle travels to node j in time t_{ij}, and naturally the vehicle will arrive at node j not before time $T_i + s_i + t_{ij}$.

Constraint [DDS91, (21)] models capacity restrictions. The total load on a vehicle at departure from a node i is Y_i. The depot is assigned node number zero, therefore $Y_0 = 0$, because vehicles are empty when they leave the depot. On departure from any pickup node the load on the vehicle is subject to restrictions: it must at least be the amount d_i loaded on at the pickup node, it can at most be the capacity D of the vehicle (in this model all vehicles have the same capacity D).

In an object-oriented software design model there will probably not be a large matrix of decision variables X_{ij}^v, instead there will be classes and objects representing the network (e.g. locations), vehicles, routes, and schedules. A corresponding UML class diagram is shown in Figure 1.

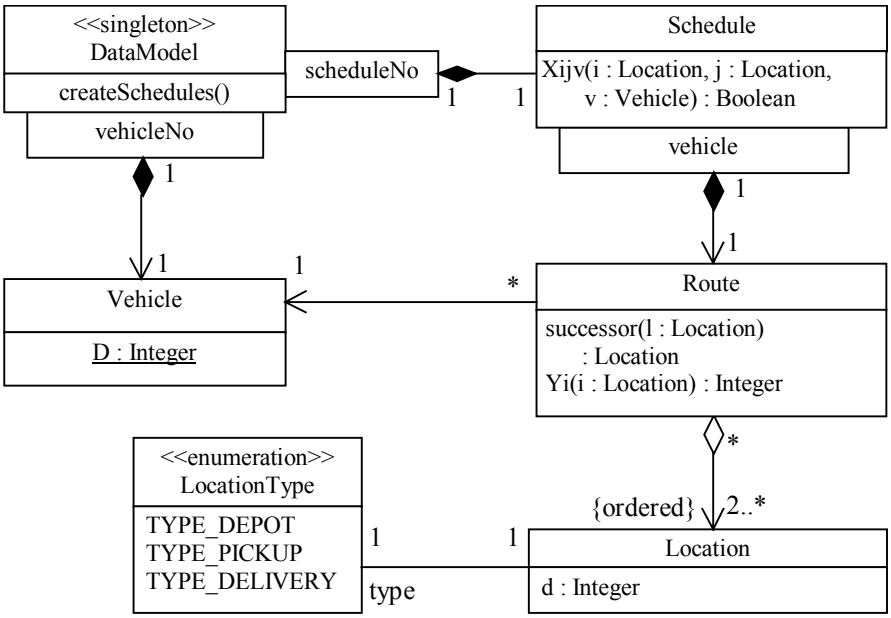

Figure 1: UML class diagram for schedules

The central class in Figure 1 is DataModel, and the only instance of DataModel represents the particular PDPTW that is to be solved. The operation createSchedules() is invoked to calculate alternative schedules according to some algorithm. To this end the operation creates a network of objects that is subject to the constraints of the PDPTW.

We use a two step approach to integrate these constraints into the software design model using OCL. In the first step the variables from the mathematical model are modelled by corresponding query operations. In the second step we use these operations to express the constraints.

Class Schedule in Figure 1 has a query operation Xijv(...) that corresponds to the decision variables X_{ij}^{v} in the mathematical model. Using OCL it is surprisingly simple to specify this operation in terms of the software design model:

```
context Schedule::Xijv(i : Location, j : Location,
      v : Vehicle) : Boolean
post: result = route→exists(r : Route
         | r.location→exists(l : Location
            | l = i and r.successor(l) = j))
```

Given the query operation Xijv(...) it is straightforward to integrate constraint [DDS91, (6)] into the software design model as a postcondition of a method createSchedules():

```
context DataModel::createSchedules()
post: let P_plus = Location.allInstances→select(
         type = #TYPE_PICKUP)
      in
   Schedule.allInstances→forAll(s : Schedule
      | P_plus→forAll(i : Location
         | Vehicle.allInstances→iterate(
               v : Vehicle; sum_1 : Integer = 0
            | sum_1 + Location.allInstances→iterate(
                  j : Location; sum_2 : Integer = 0
               | if s.Xijv(i, j, v)
                     then sum_2 + 1 else sum_2
                 endif)))
      = 1)
```

The above two step approach can be applied to other constraints like [DDS91, (12)] and [DDS91, (21)] as well. As an example this is now demonstrated in the case of [DDS91, (21)]. First the load variable Y_i can be expressed in terms of the software design model:

```
context Route::Yi(i : Location) : Integer
pre: self→contains(i)
post: let subseq = location→iterate(
        l_1; seq : Sequence(Location) = Sequence{}
        | if seq→contains(i)
             then seq else seq→append(l_1)
          endif)
      in
```

```
result = subseq→iterate(
   l_2; load : Integer = 0
   | if l_2.type = #TYPE_PICKUP
        then load + l_2.d
        else load - l_2.d
     endif)
```

The mathematical constraint [DDS91, (21)] can now be translated into the following OCL postcondition:

```
context DataModel::createSchedules()
post: let P_plus = Location.allInstances→select(
         type = #TYPE_PICKUP)
      in
      P_plus→forAll(i : Location
        | Route.allInstances→select(r_1 : Route
             | r_1.includes(i))
          →forAll(r_2 : Route
             | i.d <= r_2.Yi(i)
               and r_2.Yi(i) <= Vehicle.D))
```

Naturally a software design model becomes more formal if constraints from a mathematical model are integrated. This has had a remarkably beneficial effect on our work, because formal notations "work largely by making you think very hard about the system you propose to build" [Ha90, p. 19]. In fact the use of OCL exposed a number of shortcomings in the software design model during our work. If a formal notation is used to specify a software design then problems like inconsistencies, abiguities, and vaguenesses often become visible. The formal notation makes communication among software developers clearer, and this supports efficient discussions about designs. We have found these discussions to be a crucial factor for error detection as well as the improvement of our software design.

4 Implementation of OCL in Executable Code

From a paradigmatical point of view OCL can be counted among the logical paradigm of programming (cf. [Lo93]). Therefore problems may arise if OCL constraints are implemented using a programming language that follows the imperative paradigm, because OCL does not define a flow of control [WK99a]. We have implemented a prototype in Java, to study aspects of the transition from OCL to executable code.

The collection types defined in OCL are very powerful, and interestingly they are quite similar to the corresponding container interfaces in Java's Collection Framework. Given that, it is often straightforward to transform OCL assertions into Java code. Iterators are one of the very powerful concepts of OCL, and the Java Collection Framework provides useful realisations which facilitate the implementation of OCL constraints.

It might be a tempting idea to generate executable code from OCL assertions. This might be reasonable in some cases, but often we found that this would result in prohibitively inefficient executable code. The reason for this is that OCL assertions are often

formulated with the human reader in mind. As is well-known from logical programming, the most elegant, concise, and comprehensible formulation of a problem is often a very inefficient formulation for execution by a machine (cf. [MS98]). Therefore in many cases where OCL assertions could have been translated into executable code in a mechanical fashion this turned out to be undesirable because of performance considerations. The following OCL class invariant illustrates this case:

```
context Route inv:
location→forAll(type <> #TYPE_DEPOT)
```

This simple OCL assertion expresses that in a certain data structure there must not be any object with the property denoted "type" set to the value "#TYPE_DEPOT". Here the OCL iterator operation forAll(...) is used to iterate over the full collection any time the invariant is checked. As invariants have to be ensured after every change in the state of an object, this would result in a prohibitive run-time overhead. Therefore the implementation of this invariant in executable code followed a completely different path: There were only three operations on the data structure that added elements to it, and only in these cases a precondition was added to these operations that ensured that the added element satisfied the condition "type <> #TYPE_DEPOT".

A serious problem during the implementation of OCL assertions in executable code is their placement. The Eiffel language described in [Me97] provides special keywords that standardise the placement of assertions in program code, which clearly facilitates the traceability of OCL assertions in a software product. Other languages like for example Java do not provide a mechanism for the systematic integration of code resulting from OCL. This may lead to arbitrary placement of the assertions, which can cause difficulties during maintenance.

5 Comparing OCL to Z

Z is one of the most established formal specification languages. It is a model-orientated, strongly typed language that relies on set theory, and first order predicate logic. Z is based on so-called schemas as the basic element of specification structure, and it explicitly separates state schemas, describing admissible states of a system, and operation schemas, describing admissible state transitions. Z provides an extremely powerful and complicated schema calculus. It has an unmatched theoretical foundation and is particularly suited for correctness proofs. Z uses a mathematical formula language with a complex notation and a number of special characters (cf. [Po96]).

OCL is a model orientated, typed specification language too. But unlike Z it does not have a strong mathematical foundation. It does not use a complicated notation and it does not rely on special character sets. Any OCL expression is associated with a model element from the underlying UML model as context, and therefore the vagueness of UML translates into some degree of informality on the side of OCL assertions. For example it is perfectly legal to use an operation from the underlying UML model in an OCL expression, even if there is no formal definition of the semantics of the operation. In cases like that, OCL allows to rely on the intuition of the modeler, whereas a formal language like Z precludes such a situation completely.

Therefore OCL and Z address different groups of software specifiers. Z has emerged in academic circles and enforces strict formality, which makes it suitable for formal reasoning about systems. OCL on the other hand has been influenced by practitioners and offers a variable degree of formality. That allows to use OCL in practice where many modelers do not have a formal education in computer science or mathematics.

6 Tool Support

OCL has only recently become popular, and therefore tool support is limited as yet. We tested the free OCL Parser (Release 0.3, available from IBM at http://www.ibm.com) developed by Jos Warmer (one of the autors of [WK99a] and [WK99b]). At the time of writing the parser does only cover some parts of OCL, and it should be regarded as a learning tool rather than a productive tool. OCL is a typed language, and the OCL parser is mainly restricted to type checking, although it does not perform complete type checking yet. In particular the OCL parser does not do precondition and postcondition checking for operations, it can handle class invariants only.

The input for the OCL parser must be specified as a text file in a description language special to the OCL parser. This language is simple and straightforward, but it is not standardised and therefore none of the leading UML drawing applications (e.g. Rational Rose 4.0 or Together C++ 2.3) has an option to generate these files at present. Writing up these files manually is obviously cumbersome, error-prone, and inefficient.

7 Conclusion

In this paper we have assessed the applicability of the OCL to VRS software development. In that special application domain the established way to specify requirements is a formal mathematical model. OCL allows to integrate constraints from this mathematical model into an object-oriented software design model in a concise and seamless manner. This is clearly an important step toward efficient requirements tracing which plays a crucial role in the entire software life cycle. Formality in a software design model makes it amenable to discussion, and efficient discussions among software developers are a key factor for error detection and the improvement of software designs.

We think that the use of OCL in commercial software development can be beneficial if three conditions are met. First, formality should be a requirement (this only applies to a fraction of all software projects). Second, the use of the graphical UML should be an established practice (many commercial software developers have no idea of UML at present). Third, there should be an agreement among team members to use a more formal language (often software developers resent the use of formal techniques).

In cases where the above conditions are met we think that OCL can have a positive impact on software development in a commercial setting. The reasons for this are: First, OCL is simple, and simplicity is a major criterion in commercial software development. Second, OCL was clearly developed with the human user in mind, it is very convenient to use and does not rely on complicated notations or special character sets. Third, OCL enriches the established UML in a modular fashion, therefore it is the natural upgrade path for the users of the graphical notation. Fourth, OCL allows a flexible degree of

formality, and that makes it particularly appealing to practitioners, because it supports a "learn as you go" approach. Fifth, OCL incorporates concepts that are already familiar to users of object-oriented programming languages, e.g. containers and iterators.

Difficulties may arise during the implementation of OCL assertions in an imperative language. These problems can be overcome in practice by a pragmatic attitude. The problems could be alleviated by using a particularly suitable implementation language like Eiffel, but this is not a feasible option in the commercial field.

When it comes to tool support then OCL lags far behind more traditional specification languages like Z. The adoption of OCL in commercial software development will crucially depend on the support provided for it by the leading UML drawing tools.

References

[DDS91] Dumas, Y., Desrosiers, J., Soumis, F.: The pickup and delivery problem with time windows. European Journal of Operational Research, Vol. 54, 1991, pp. 7-22.

[Ha90] Hall, A.: Seven Myths of Formal Methods. IEEE Software, Vol. 7, No. 5, 1990, pp. 11-19.

[Ja98] Jarke, M.: Requirements Tracing. Communications of the ACM, Vol. 41, No. 12, 1998, pp. 32-36.

[Lo93] Louden, K. C.: Programming Languages - Principle and Practice. PWS Publishing Company 1993.

[MS98] Marriot, K.; Stuckey, P. J.: Programming with Constraints - An Introduction. MIT Press 1998.

[Me97] Meyer, B.: Object-Oriented Software Construction. Prentice Hall 1997.

[OMG00] Object Management Group (Editor): OMG Unified Modeling Language Specification, Version OMG-UML V 1.3, March 2000. Object Management Group, Needham (USA) 2000, http://www.omg.org.

[Po96] Potter, B.; Sinclair, J.; Till, D.: An Introduction to Formal Specification and Z. Prentice Hall 1996.

[Pr00] Priestley, M.: Practical Object-Oriented Design with UML. McGraw-Hill 2000.

[So87] Solomon, M. M.: Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. Operations Research, Vol. 35, No. 2, 1987, pp. 254-265.

[WK99a] Warmer, J.; Kleppe, A.: The Object Constraint Language - Precise Modeling with UML. Addison Wesley Longman 1999.

[WK99b] Warmer, J.; Kleppe, A.: OCL - The Constraint Language of the UML. The Journal of Object-Oriented Programming, Vol. 12, 1999, pp. 10-13 and p. 28.