CS335: Model Driven Engineering & Object Constraint Language

Overview

Hao Wu

Computer Science Department,
National University of Ireland, Maynooth

Email: haowu@cs.nuim.ie

March 12, 2018

OCL

The **Object Constraint Language** (OCL) is a modeling language with which you can build software models.

It is defined as a standard "add-on" to the **Unified Modeling Language** (UML), the **Object Management Group** (OMG) standard for object oriented analysis and design.

Every expression written in OCL relies on the types (i.e., the classes, interfaces, and so on) that are defined in the UML diagrams. The use of OCL therefore includes the use of at least some aspects of UML.

Model Driven Architecture

The Model Driven Architecture (MDA) is gradually becoming an important aspect of software development. Many tools are, or at least claim to be, MDA-compliant, but what exactly is MDA?

MDA is a framework being built under supervision of the Object Management Group (OMG). It defines how models defined in one language can be transformed into models in other languages.

An example of a transformation is the generation of a database schema from a UML model. UML being the source language and SQL being the target language.

MDA Benefits

- Portability: increasing application re-use and reducing the cost and complexity of application development and management, now and in the future.
- Productivity: by enabling developers, designers, and system administrators to use languages and concepts they are comfortable with, while still supporting seamless communication and integration across the teams.
- Cross-platform interoperability: using rigorous methods to guarantee that standard based on multiple implementation technologies all implement identical business functions.
- Easier maintenance and documentation: MDA implies that much of the information about application must be incorporated in the *Platform Independent Model* (PIM).

The Silver Bullet?

When explaining the MDA to software developers, we often get a skeptical response: "this can never work. You cannot generate a complete working program from a model. You will always need to adjust the code." Is MDA just promising another silver bullet?

MDA is still in its infancy, it shows the potential to radically change the way we develop software. We likely witness the birth of a paradigm shift: and in the near future, software development will shift this focus from code to models.

Modeling Maturity Levels

Traditionally, there has been a gap between the model and the system. The model is used as a plan, as brainstorm material, or as documentation, but the system is the real thing. Often, the detailed software code strays a long way from the original model.

On every modeling level, this gap is reduced. As one climbs to a higher maturity level, the term *programming* gets a new meaning. At a higher maturity level, modeling and programming become almost the same.

Level 0: No Specification

One simply gets an idea about what to develop, and talks about it without ever writing anything down. The characteristics of this level are as follows:

- There are often conflicting views among developers, and between developers and users.
- This manner of working is suitable for all small applications; larger and more complex applications need some form of design before coding.
- It is impossible to understand the code if the coders leave (they always do).
- Many choices are made by the coders in an ad hoc fashion.

Level 1: Textual

At this level, the specification of the software is written down in one or more natural language documents. These may be more or less formal, using numbering for every requirement or every system function or not, large or small, an overview or very detailed, depending on taste. The characteristics of this level are as follows:

- The specification is ambiguous, because natural language inherently is.
- The coder makes business decisions based on his or her personal interpretation of text(s).
- It is impossible to keep the specification up to date after changing the code.

Level 2: Text with Diagrams

At this level, the software is provided by one or more natural language documents augmented with several high-level diagrams to explain the overall architecture and/or some complex details. The characteristics of this level are as follows:

- The text still specifies the system, but it is easier to understand because of the diagrams.
- All characteristics of level 1 are still present.

Level 3: Models with Text

At this level, a set of model are introduced. The characteristics of this level are as follows:

- The diagrams or formal texts are real representations of the software.
- The transition of model to code is mostly manual.
- It is still impossible to keep the specification up to date after changing the code.
- The coder still makes business decisions, but these have less influence on the architecture of the system.

Level 4: Precise Models

A model, meaning a consistent and coherent set of texts and/or diagrams with a very specific and well-defined meaning, specifies the software. The models at this level are precise enough to have a direct link with the actual code. However, they have a different level of abstraction.

- Coders do not make business decisions anymore.
- Keeping models and code up to date is essential and easy.
- Iterative and incremental development are facilitated by the direct transformation from model to code.

Level 5: Models Only

Model is a complete, consistent, detailed, and precise description of the system. At this level, models are good enough to enable complete code generation. No adjustments needs to be made to the resulting code. Software developers can rely on the model-to-code generation in the same way coders today rely on their compilers.

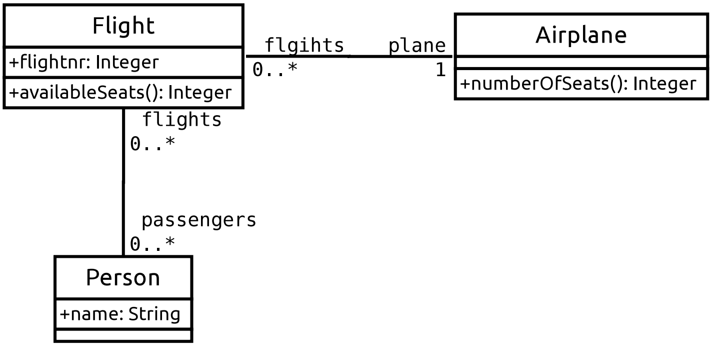
Note that this level has not been realized yet anywhere in the world. This is the future technology, unfortunately. Still, it is good to recognize what our ultimate goal is.

Why combine UML and OCL?

The combination of UML and OCL offers the best of both worlds to the software developer. A large number of different diagrams, together with expressions written in OCL, can be used to specify models. Note that to obtain a complete model, both the diagrams and OCL expressions are necessary.

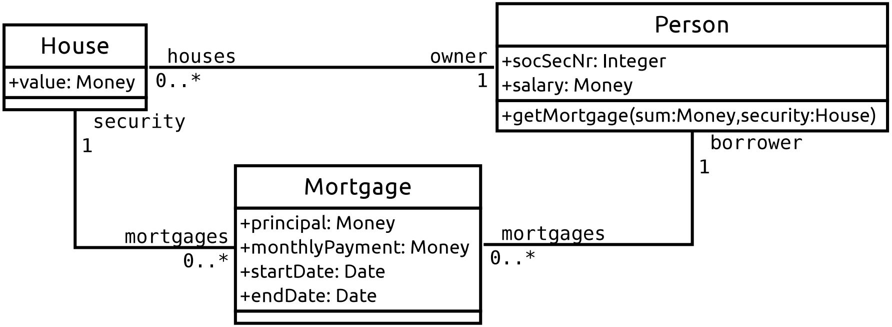
Without OCL expressions, the model would be severely underspecified; without UML diagrams, the OCL expressions would refer to non-existing model elements, as there is no way in OCL to specify classes and associations.

An Exmaple



```
context Flight
inv: passengers->size()<=plane.numberOfSeats
```

Mortgage System



Some Rules

Any human reader of the model (previous slide) will undoubtedly assume that a number of rules must apply to this model.

1. A person may have a mortgage on a house only if that house is owned by him or herself; one cannot obtain a mortgage on the house of one's neighbour or friend.
2. The start date for any mortgage must be before the end date.
3. The social security number of all persons must be unique.
4. A new mortgage will be allowed only when the person's income is sufficient.
5. A new mortgage will be allowed only when the countervalue of the house is sufficient.

However, the diagram does not show this information; nor is there any way in which the diagrams might express these rules.

OCL Rules

```
context Mortgage
inv: security.owner = borrower

context Mortgage
inv: startDate < endDate

context Person
inv: Person:allInstances()->isUnique(socSecNr)

context Person::getMortgage(sum: Money, security: House)
pre: self.mortgage.monthlyPayment->sum()<=self.salary*0.30

context Person::getMortgage(sum: Money, security: House)
pre: security.value >= security.mortgages.principal->sum()
```

Value Added by OCL

It is essential to include these rules as OCL expressions in the model for a number of reasons. No misunderstanding occurs when humans read the model.

More importantly, errors can be found in an early stage of development, when fixing a fault is relatively cheap. The intended meaning of the analyst who builds the model is clear to the programmers who will implement the model.

When the model is not read by humans, but instead is used as input to an automated system, the use of OCL becomes even more important. Tools can be used for generating simulations and tests, for checking consistency, for generating code and so on. This type of work most people would gradually leave to a computer, if it would and could be done properly.

Characteristics of OCL

1. Both query and constraint language. (OCL has the same capabilities as SQL). However, neither OCL nor SQL is a constraint language. OCL is a constraint and query language at the same time.
2. Mathematical foundation, but no mathematical symbols. OCL is based on mathematical set theory and predicate logic, and it has a formal mathematical semantics.
3. Strongly typed language. Because most models are not executed directly, most OCL expressions will be written while no executable version of the system exits.
4. Declarative language. An OCL expression simply states **what** should be done, but not **how**. The model can make decisions at a high level of abstraction, without going into detail about how something be calculated. For example, the body expression of the operation *availableSeats*().

Summary

OCL is a **language** that can express additional and necessary information about the models and other artifacts used in object-oriented modeling, and should be used in conjunction with UML diagrammatic models.

OCL is a **precise**, **unambiguous** language that is easy for people who are not mathematicians or computer scientist to understand.

OCL is a **declarative**, **side-effects-free** language; that is, the state of a system does not change because of an OCL expression. More importantly, a modeler can specify in OCL exactly what is meant, without restricting the implementation of the system that is being modeled.