TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

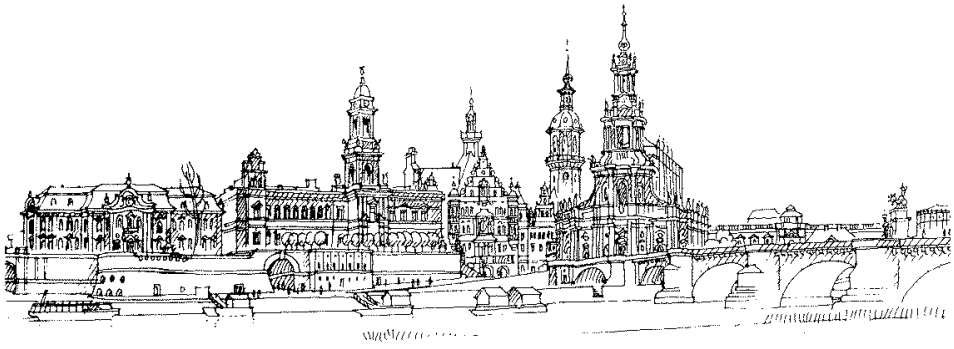
Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI06-04-Sept. 2006

Birgit Demuth, Dan Chiorean,
Martin Gogolla, Jos Warmer (eds.)

Institut für Software- und Multimediatechnik

**OCL for (Meta-)Models
in Multiple Application Domains**



Technische Universität Dresden
Fakultät Informatik
D−01062 Dresden
Germany
URL: http://www.inf.tu−dresden.de/

Birgit Demuth, Dan Chiorean,
Martin Gogolla, Jos Warmer (eds.)

OCL for (Meta-) Models in Multiple Application Domain

Workshop co-located with MoDELS 2006: 9th International Conference on Model-Driven Engineering Languages and Systems (formerly the UML series of conferences)

October 1 - 6, 2006 Genova, Italy

This proceedings have been funded by the European Commission and by the Swiss State Secretariat for Education and Research within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net).

Organization

Organizers

Dan Chiorean (Romania)
Birgit Demuth (Germany)
Martin Gogolla (Germany)
Jos Warmer (The Netherlands)

Program Committee

Thomas Baar (Switzerland)
Jordi Cabot (Spain)
Dan Chiorean (Romania)
Tony Clark (United Kingdom)
Birgit Demuth (Germany)
Andy Evans (United Kingdom)
Robert France (U.S.A.)
Martin Gogolla (Germany)
Heinrich Hussmann (Germany)
Marcel Kias (Norway)
Richard Mitchell (U.S.A.)
Octavian Patrascoiu (United Kingdom)
Mark Richters (Germany)
Shane Sendall (Switzerland)
Peter Schmitt (Germany)
Jos Warmer (The Netherlands)
Burkhart Wolff (Switzerland)

Preface

This Technical Report contains the final versions of papers accepted at the workshop *OCL for (Meta-)Models in Multiple Application Domains*, held in Genua (Italy), October 2, 2006. The workshop was organized as a satellite event of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006). It continues the series of OCL workshops held at previous UML/MoDELS conferences: York (2000), Toronto (2001), San Francisco (2003), Lisbon (2004) and Montego Bay (2005). Similar to previous OCL workshops, the majority of papers were proposed by researchers coming from the academia. Apart from the above mentioned workshops, three accepted papers were proposed by researchers affiliated to IBM and SAP, two of most influential worldwide software companies. Hoping that this represents an encouraging sign of a future enhanced use of the OCL in the industry.

The 18 accepted papers cover a large spectrum of OCL related topics. They reflect research contributions and experience reports about using OCL for models and meta models in multiple application domains. The papers are divided into four sections concerning new applications, model transformations pertaining to the MoDELS/UML 2006 conference as well as implementation and language issues.

In the section *Applications* there are five contributions including new paradigms and technologies. The paper *Customer Validation of Formal Contracts* by Heldal and Johannisson deals "customer-readable" specifications for systems modeled with UML and OCL. Kolovos, Paige and Polack show in their paper *Towards Using OCL for Instance-Level Queries in Domain Specific Languages* how to align OCL querying and navigation facilities with arbitrary Domain Specific Languages (DSLs). In the paper *OCL-based Validation of a Railway Domain Profile* Berkenkötter uses OCL to validate models of a railway domain profile and the profile itself. Opoka and Lenz built a model assessment framework based on EMF and show in their experience report *Use of OCL in a Model Assessment Framework: An Experience report* that OCL 2.0 is expressive enough to be applied as a query language for model analysis. In the paper *Rigorous Business Process Modeling with OCL* the “Business-IT gap" issue, a well-known problem in business process modelling, is investigated. Takemura and Tetsuo Tamai enrich activity diagrams by (extended) OCL constraints to model business processes rigorous.

Some of the OCLApps2006 papers deal with model transformations that are contained in the section *Model transformations*. Milanovic et al focus in their paper *On Interchanging Between OWL/SWRL and UML/OCL* on the reconciliation of OCL and Semantic Web languages. In the paper *Realizing UML Model Transformations with USE,* Büttner and Bauerdick enrich the USE specification language with imperative elements to realize class diagram transformations. Wahler, Koehler and Brucker form the notion of *Model-Driven Constraint Engineering* in their paper with the same title. The idea is to define constraint patterns, integrate them into the UML metamodel and transform them into platform-specific constraints by model transformation.

In the section *Implementations* several authors deal with a better implementation of OCL support in tools. In the paper *OCL support in an industrial environment* Altenhofen, Hettel and Kusterer describe their method of impact analysis to reduce the number of necessary (re-)evaluations of OCL constraints when the underlying model changes. Stölzel, Zschaler and Geiger discuss in the paper *Integrating OCL and Model Transformations in Fujaba* the integration of the Dresden OCL Toolkit into the Fujaba Tool Suite. This adds OCL support

both for class diagrams and for model transformations. Mezei, Levendovszky and Charaf present in the paper *Restrictions for OCL constraint optimization algorithms* an optimization algorithm for the evaluation of OCL constraints used in graph rewriting based model transformations. The paper *An MDA Framework Supporting OCL* by Brucker, Doser and Wolff describes a rather complete tool chain for processing UML/OCL specifications, including a proof environment and flexible code generation. Amelunxen and Schürr explain in the paper *On OCL as part of the metamodeling framework MOFLON* the role of OCL in the metamodeling framework MOFLON, a framework designed by the authors to support the definition of domain specific languages with MOF, OCL and graph transformations.

In the section *Language issues* there are multiple proposals to improve the syntax and semantics of the language. Cabot identifies in the paper *Ambiguity issues in OCL postconditions* common ambiguities appearing in OCL postconditions and provides a default interpretation for each of them in order to improve the usefulness of declarative specifications. Akehurst, Howells and McDonald-Maier sketch in the paper *UML/OCL - Detaching the Standard Library* some ideas for detaching certain aspects of the OCL language such as types and operations from its current definition. The goal is to facilitate support for different modeling languages and implementations. In the paper *Semantic Issues of OCL: Past, Present, and Future* Brucker, Doser and Wolff summarize the most important results of a formalization of OCL and proceed to make suggestions on how OCL may be improved in order to have a cleaner formal semantics. The paper *Improving the OCL Semantics Definition by Applying Dynamic Meta Modeling and Design Patterns* by Chiaradía and Pons try to improve the definition of the OCL semantics (in particular the Expression-Evaluation package) by means of applying the visitor pattern and the Dynamic Meta-Modeling technique. Süß proposes in the paper *Sugar for OCL* three shorthand notations for representing OCL in the Latex, HTML, and Unicode encoding systems.

Finally, we would like to thank all members of the program committee for their dedication to writing reviews and for useful suggestions to improve the submitted papers. We were very pleased about the numerous contributions to this year's OCL workshop. Thank to the authors of all submitted papers who made this workshop possible. May the workshop presentations and discussions inspire all workshop attendees to enhance the application of OCL in industrial environments, the tool support for OCL, the scientific foundation and, last but not least, the language OCL itself.

September 2006

Birgit Demuth
Dan Chiorean
Martin Gogolla
Jos Warmer

Table of Contents

Applications

Model Transformations

Implementations

Language Issues

Customer Validation of Formal Contracts

Rogardt Heldal and Kristofer Johannisson

Chalmers University of Technology
Gothenburg, Sweden
[heldal|krijo]@cs.chalmers.se

Abstract. This paper shows how to write formal OCL contracts for system operations in such way that a translation to natural language (a subset of English), understandable by a customer, can be obtained automatically. To achieve natural language text understandable by a customer we use the vocabulary of the problem domain when writing formal contracts for system operations. The benefits of our approach are that we increase the precision of the model by using formal specifications, and that a customer is able to validate (by viewing the natural language rendering) if a contract actually describes the behavior desired from the system. Without validation of this kind there is generally no guarantee that the formal specification states the correct properties.

1 Introduction

Large programs need specifications. These specifications might be used in different contexts: for software developers to support the implementation of the software, for testers to understand the required behavior of the software, for customers to validate the correctness of the system, and for users of the software to understand the behavior of the system.

It is important that the specifications are of high quality to avoid problems such as ambiguity and under-specification. To that end, several formal languages have been developed to write more precise specifications [1, 4, 5]. The nature of these languages forces one to be more precise than when using natural language to specify behavior of programs. The problem is, however, that not everyone involved in the software process — for example the customers — can be expected to understand these formal languages. So, the customer cannot, at least easily, validate whether a formal specification states the correct behavior or not. There is little point in being precise if the specification states wrong properties. This is the problem we deal with in this paper.

We consider formal contracts similar to those in Eiffel [11] by attaching OCL (Object Constraint Language)[12] constraints to *system operations* in UML [12] class diagrams. The constraints specify the signature of an operation together with its pre- and post-conditions. We previously developed a tool[3] which translates OCL constrains to natural language text (a subset of English). But removing the overhead of OCL does not necessary make the natural language text

understandable for customer and users. The text might contain too many design and implementation details and is therefore more suited for designers and implementers.

This paper is about controlling the vocabulary used in OCL pre- and post-conditions for system operations, in such a way that natural language text produced by our tool can be read by customers and users having domain knowledge, but not necessarily computer science knowledge. Not only does our tool make translation understandable for customers, users, and domain experts, but it also permits the tool to be used earlier in a software development process than previously [3]. With respect to our previous work in [9, 3], the contribution of this paper is not to improve the tool itself, but rather a new use of the tool.

In our previous work [3] the focus was on the quality of the translation for any type of OCL contracts. In this work we restrict ourself to contracts for system operations, which represent the external interface of the system. The reason for only considering these operations is that customers and users are mostly interested in these operations, and not in the internals of the system. It is crucial that the contracts for system operations capture the behavior customers want. It is impossible to build a system correctly if one does not know its expected behavior. It is well known that requirement deficiencies are the prime source of project failures [7].

We introduce the notion of *abstract contracts*, which abstract away from implementation details by using *problem domain models* instead of class diagrams for system operations contracts. Both problem domain models and class diagrams can be represented by UML class diagrams, but they differ in an important way: the vocabulary of domain models should be fully understood by the customer but it is not a requirement for class diagrams used in the design phase. Domain models restrict OCL constraints to a vocabulary, which should be completely understood by the customer. Translating these formal contracts using our tool does not only remove OCL, but also guarantees that the natural language text obtained contains vocabulary of the domain model — the vocabulary of the customers, users, and domain experts. So the only requirement for reading these specifications are domain knowledge which permits validation of the abstract formal contracts without having to read OCL — the customer only needs to read the natural language text produced by our tool.

There are further benefits of our work going beyond the validation of formal specifications. The natural text produced can be used as a documentation of the system. Since the text is generated from formal specifications, even the natural language text will be in some sense formal, but of course readable for customers. So, the benefit is more precise natural language specifications avoiding ambiguity and under-specification. Furthermore, if the formal specifications change, one can just produce new natural language text, avoiding the problem of synchronizing formal and informal specifications.

One important lesson to learn from our work is that if the translation being for most parts compositional, as in our case, from formal to natural language the choice of vocabulary of the formal specification becomes crucial in controlling

the vocabulary of the natural text produced. Customer- and user-understandable natural text does not come for free. In our case the trick was to use the domain model.

The key contributions: (1) We create a formal, machine-transformable model early in the software engineering process (2) The model can be validated by customers, since the OCL contracts can be translated into understandable natural language. (3) By automatically translating OCL to natural language and using OCL as a single source, we avoid the problem of synchronizing formal specifications with natural language text describing customer requirements. (4) The natural text produced can be used as part of the documentations of systems.

Paper Overview. Some background on the translation tool and domain models is given in Sect. 2 and 3. We then discuss system operations, contracts and the vocabulary of the domain model in Sect. 4, 5 and 6. Finally, Sect. 7 describes related work and we conclude in Sect. 8.

2 From OCL To Natural Language

We have developed a tool for linking specifications in OCL to natural language specifications. It has been previously described in [9], where we give basic motivation and design principles, and in [3], where we conduct a case study. It is based on the Grammatical Framework (GF) [13], and is being integrated into the KeY system [2]. The basic idea of the tool is to define an abstract representation (an abstract syntax) of "specifications" and to relate this representation to both OCL and English using a GF grammar. The GF system then allows us to translate between OCL and English using the abstract syntax as an interlingua. We can therefore always keep OCL and English in sync.

Given a UML model, the tool provides two basic functionalities: (1) Automatic translation of OCL specifications into English. (2) A multilingual, syntax-directed editor which allows editing of OCL and English specifications in parallel. The input to the translator is an OCL specification, along with a description of the UML model (a domain model or class diagram) in question. The output is English text, formatted in HTML or LaTeX. In the editor, OCL and English are kept in sync since the editing takes place on the level of abstract syntax.

In previous work [3] we did not consider the style of formal abstract contracts. These contracts contained design and implementation details, and we refer to them as *concrete contracts*, for an example see Fig. 1. This figure shows parts of the OCL specification for the operation *check* of the class *OwnerPIN*, a class in the Java Card API. Java Card [17] is a subset of Java, tailored to smart cards and similar devices, which comes with its own API. The operation *check* checks whether a given PIN matches the PIN on a smart card, keeps track of the number of times you have entered an incorrect PIN, and so on.

Translating concrete contracts to natural language text removes the overhead of having to understand OCL, but they still contain design and implementation details, see Fig. 2. This figure shows the natural language translation provided

```
context OwnerPIN::check(pin: Sequence(Integer),
         offset: Integer, length: Integer): Boolean
post: (self.tryCounter > 0 and not (pin <> null and offset >= 0
            and length >= 0 and offset+length <= pin->size()
            and Util.arrayCompare(self.pin, 0, pin, offset, length) = 0)
      ) implies (not self.isValidated()
            and self.tryCounter = tryCounter@pre-1 and
            (( not excThrown(java::lang::Exception) and result = false)
               or excThrown(java::lang::NullPointerException) or
               excThrown(java::lang::ArrayIndexOutOfBoundsException)))
```

Fig. 1. Design Level OCL Contract

by our system. The translation will make sense only to a reader who already has an understanding of the Java Card API classes. He or she must be familiar with the use of byte arrays and their representation in OCL, as well as with various Java Card exceptions.

For the operation **check (pin : Sequence(Integer) , offset : Integer , length : Integer) : Boolean** of the class **javacard::framework::OwnerPIN** ,
the following post-condition should hold :

- if the try counter is greater than 0 and at least one of the following conditions is not true
  - *pin* is not equal to null
  - *offset* and *length* are at least 0
  - *offset* plus *length* is at most the size of *pin*
  - the query `arrayCompare ( the pin , 0 , pin , offset , length )` on `Util` is equal to 0

  then this implies that the following conditions are true
  - this own er PIN is not validated
  - the try counter is decremented by 1
  - at least one of the following conditions is true
    - `Exception` is not thrown and the result is equal to false
    - `NullPointerException` is thrown
    - `ArrayIndexOutOfBoundsException` is thrown

Fig. 2. Design Level Natural Language Contract

In summary, concrete contracts, in formal or natural language, are not generally understandable to customers, since customers cannot be expected to be familiar with design and implementation matters.

If we compare the natural language text in Fig. 2 to the original OCL contract in Fig. 1, we can note that they both share roughly the same structure

and vocabulary. This is a consequence of our translation being (for most parts) compositional, using the same abstract syntax for both OCL and natural language. If an OCL contract uses the vocabulary of a UML class diagram (design phase), which is not normally understandable to a customer, the natural language translation will not be understandable either. This is fine since concrete contracts are meant to be read and understood by designers and implementers but not customers and users.

The goal of this work is to obtain specifications understandable by customers, users, and domain experts not trained in computer science. We show how domain models can be used for this purpose, but first we will have closer look at what we mean by a domain model.

3 Domain Models

In this section we provide background on domain models used in our work. The problem domain models we use are as in the book of Larman [10]. A problem domain model is a visualization of concepts in a real-life domain of interest without behavior, not software classes such as Java, C++, or C# classes.

UML does not offer separate notation for domain models, but a restricted form of UML class diagrams can be used. For example, the operation compartment of UML classes are not needed. On the other hand, the benefit is that this permits OCL constraints to be written over the domain models. As a running example we will consider an ATM system. The concepts involved in such a system would include *Card*, *Customer*, *Account*, and *Bank*, as shown in Fig. 3. In general, a banking application would require more concepts and attributes, but for our purposes this domain model is sufficient. To make our example simpler, we assume that a customer can never have more than one account.

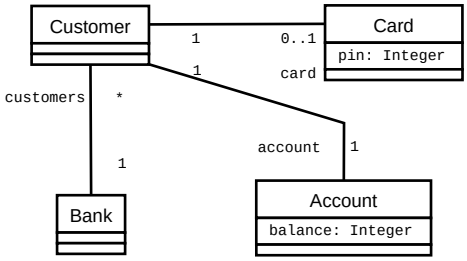


Fig. 3. Domain model of an ATM machine

Concepts can have attributes. For example, a *Card* has a PIN code, and an *Account* has a balance. Furthermore, concepts can stand in relationship to each other; in our example a *Card* is associated with *Customer*. With the help of multiplicity annotations, one can describe constraints on how many instances of

one concept are related to another. For example, one can express the constraint that each card can be linked to only one customer.

There are no hard and fast rules of how to choose the correct abstraction level for domain models. But there is one rule which should never be broken: whatever abstraction level one chooses, the domain model should always be understandable for customers, users, and domain experts. Of course, the more concrete and detailed domain model the more concrete contracts can be written. Yet, a domain model should never be so concrete and detailed that the customer does not understand it anymore.

Before considering how domain models permit to create formal contracts rather early in the development process we will have a closer look at system operations.

4 System Operations

System operations are the operations that deal with the events coming from outside the system. For information about how to obtain system operations from use cases we refer to [15]. From a customer's point of view system operations are the important operations: they define the functionality of the system. To exemplify we consider three operations for our ATM system: *identify*, *authenticate*, and *withdraw*, as seen in Fig. 4. We collect the system operations in a class *ATMController*, further explained in Sect 5.

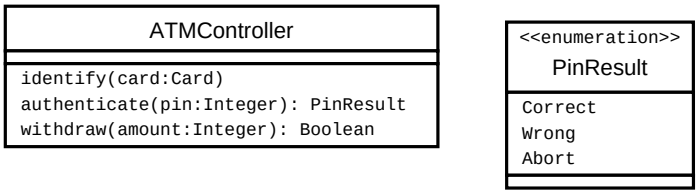


Fig. 4. System Operations for an ATM

The identity of a customer is given by presenting a card, using the operation *identify*, the operation *authenticate* ensures that the card is used by the authorized person, and *withdraw* is used to retrieve money from the ATM machine. We introduce an enumeration type to indicate the result of *authenticate*: either the pin is accepted (*Correct*), or it is wrong but the user is given another chance to enter the correct code (*Wrong*), or the wrong code has been given too many times (*Abort*). The result of *withdraw* is a boolean, indicating whether there was enough money in the account or not. In next section we will consider how to obtain contracts for these system operations.

5 Abstract Formal Contracts

To write OCL contracts for system operations requires a context for the operations. Our approach is to include all system operations in one UML class and associate this class to appropriate concepts of the problem domain — creating a hybrid between a class representing the system and concepts. This permits us to write OCL pre- and post-conditions for the system operations. It might be necessary to add attributes to the system class to model the state of the system not captured by the domain model. In our running ATM example, we put the system operations in a class *ATMController* and attach it to the domain model as shown in Fig. 5. We have also added two attributes *numberOfTries* and *pinAccepted* to keep track of the state.

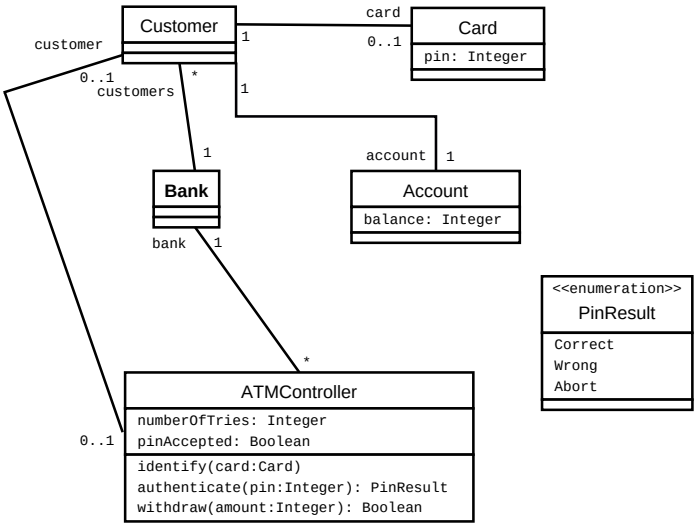


Fig. 5. The System Class Attached to the Domain Model

The UML class diagram in Fig. 5 can be viewed as the first approximation of the system to be built, but it is important to point out that it is not the final software design. In an object oriented design one will expect that some or all of the concepts of Fig. 5 to become design classes with operations and possibly more attributes. Furthermore, new classes will most often be added due to for example design patterns. The purpose of the class diagram in Fig. 5 is to permit writing formal contracts for system operations using the vocabulary of the customer.

Fig. 6 shows abstract OCL contracts for our example ATM system operations using the class diagram of Fig. 5. Let us consider the operation *identify*. This operation should initialize the state of *ATMController*: the association from

```
context ATMController::identify(card:Card)
post: customer = card.customer
      and numberOfTries = 0
      and not pinAccepted


context ATMController::authenticate(userPin:Integer) : PinResult
pre: not pinAccepted
post: numberOfTries = numberOfTries@pre + 1
      and if userPin = customer.card.pin and numberOfTries <= 3
          then pinAccepted and result = PinResult::Correct
          else not pinAccepted
               and if numberOfTries <= 3
                   then result = PinResult::Wrong
                   else result = PinResult::Abort
                   endif
          endif

context ATMController::withdraw(amount:Real) : Boolean
pre: pinAccepted
post: if (amount <= customer.account.balance)
      then customer.account.balance =
                customer.account.balance@pre - amount
           and result = true
      else customer.account.balance =
                customer.account.balance@pre
           and result = false
      endif
```

Fig. 6. OCL Abstract Contracts

ATMController to *Customer* is instantiated with the customer of the card, the number of tries is 0, and a correct pin code has not yet been entered.

Obtaining a customer from the association of the card is not enough. In addition, a guarantee that one is dealing with the right customer is desirable. According to the domain model the concept *Card* is related to a customer. So, the operation *authenticate* compares the PIN given as argument with the PIN stored in the card associated with the customer. Depending on whether the PIN argument matches the PIN on the card, and on the number times an incorrect PIN has been entered, the state of the system class *ATMController* is set appropriately.

Finally, there is *withdraw* which gives the customer money if there is enough money in the account. The return value indicates whether the withdrawal was successful or not.

Both problem domain model and system operations are found early in the development process, so abstract contracts like the one in Fig. 6 can be created

at an early stage of the development process. Even though the contracts in Fig. 6 are not readable for people not trained in formal methods, we will see that the natural language counterpart will indeed be readable.

Using the domain model in the contracts also provides a validation of the domain model, which is important since the domain model is the foundation of the system to be built. One might discover shortcomings — e.g. a missing attribute or concept — when creating the contract. In this sense, writing the contract over the domain model may improve the domain model itself.

6 The Vocabulary of the Domain Model

Fig. 7 shows what our translator produces from the OCL specification in Fig. 6. Natural language contracts should be understandable to customers not only because it has been translated from OCL to natural language, but also because it uses a vocabulary of the problem domain model.

If we compare the natural language text in Fig. 7 to the original OCL contract in Fig. 6, we can note that they both share roughly the same structure and vocabulary. Again, this is a consequence of our translation being (for most parts) compositional, using the same abstract syntax for both OCL and natural language. In comparison to the natural language text produced in Fig. 2 the text in Fig. 7 do not contain any design and implementation details.

Improving the style of the natural language text is still ongoing research. For example, we can see from Fig. 5 that fragments of the formal abstract contract can be found in the natural language text, such as $identify(card : Card)$. We have not found a way of translating this fragment better and more clear than just keeping the method signature.

It is important to point out that formal abstract contracts such as in Fig. 6 probably have to be produced by formal methods experts, but customers and users only need to consider the natural language counterpart.

In this work we are not directly considering the problem of formalization: constructing formal specifications from informal ones. However, once an OCL contract has been developed, our tool can be used to generate a natural language contract, which can then be validated against the informal specification. This can be used to improve the original informal specification as well as the formal one.

7 Related Work

The approach taken in [16, 15] is similar to ours in the sense that they also provide OCL contracts for system operations. An important difference to our work is that they do not consider customer validation of the OCL contracts, which reduces the benefit of using formal contracts at an early stage in the process. They also consider how to find the system operations based on descriptions of use cases, which we do not discuss. Our work is more focused on the domain model and the translation to natural language.

For the operation **identify (card: Card)** of the class **ATMController** ,
the following post-condition should hold:

- the following conditions are true
  - the customer is equal to the customer of *card*
  - the number of tries is equal to 0
  - the property pinAccepted does not hold

For the operation **authenticate (userPin: Integer): PinResult** of the class **ATMController** ,
given the following pre-condition:

- the property pinAccepted does not hold

then the following post-condition should hold:

- the number of tries is incremented by 1 and if *userPin* is equal to the pin of the card of the customer and the number of tries is at most 3 then:
  - the property pinAccepted holds and the result is equal to `Correct`

  otherwise:
  - the property pinAccepted does not hold and if the number of tries is at most 3 then:
    - the result is equal to `Wrong`

    otherwise:
    - the result is equal to `Abort`

For the operation **withdraw (amount: Real): Boolean** of the class **ATMController** ,
given the following pre-condition:

- the property pinAccepted holds

then the following post-condition should hold:

- if *amount* is at most the balance of the account of the customer then:
  - the balance of the account of the customer is decremented by *amount* and the result is equal to true

  otherwise:
  - the balance of the account of the customer does not change and the result is equal to false

Fig. 7. Contracts in Natural Language

The UP process [10] also makes use of contracts for system operations, which are based on the domain model. However, the contracts are informal. We believe that rather than having another informal specification, system operation contracts are a good place to become formal. Our approach could be incorporated into the UP process by replacing informal system operation contracts with formal ones. In that way one can introduce formal specification into the UP process at an early stage in the development process.

Previously, we have also been working on relating formal and informal specifications [6]. In that paper we gave a method for relating post-conditions of use cases to OCL contracts. In that paper, we did not consider different styles of writing contracts or translation to natural language text. Furthermore, we required formal method experts to use our method of relating formal and informal specifications.

The idea of producing natural language from a formal representation to enable validation by people not trained in formal languages is familiar from conceptual modelling and requirements engineering. For instance, the paper [8] presents a system for generating natural language explanations from conceptual models, and also gives an overview of related work. The basic difference to our approach is that we translate textual OCL contracts for system operations — not the domain model itself — into natural language, while [8] generates explanations from "... process-oriented and static ER-like languages...", e.g. data flow diagrams. As explained in Sect. 6, the translations we provide have the same basic structure as the OCL specifications, and are understandable to a customer because of the use of the vocabulary of the domain model. In contrast, the process of generating explanations from conceptual models in [8] involves information extraction from the model, and various strategies for presenting this information in natural language. There is also work on going in the other direction: from informal, natural language text to conceptual models, e.g. [14].

8 Conclusion and Future Work

We have presented a way of attaching system operations to a domain model which permits the creation of formal abstract contracts, which in turn can be translated to natural language understandable to a customer. The purpose is to provide a precise model at an early stage which can be validated by customers. Formal specifications require more precision than informal ones, such as informal contracts and use cases. So, the developer has to make decisions about the behavior of the system. Our approach permits the customer to validate that the decisions made in the formal specification are really what the customer wants. This is crucial to provide a good starting point for the development process.

Since we have an automatic translation from OCL to natural language, we can always keep formal and natural language contracts synchronized. This is in general considered a very hard task. We have also identified the importance of using domain models when writing contracts, in particular with respect to the readability of natural language translation of the contracts.

As future work, we want to look at how to relate our work to Model Driven Architecture (MDA) [12]. Maybe our abstract contract could be a good starting point for MDA transformations. At some point models need to be related to informal specifications.

In addition, we plan to test how real customers react to our generated texts. The vocabulary should be known to the customers since the generated text contains the vocabulary of the domain model. However, the style of the generated text might be confusing.

We also plan to further investigate the quality of the natural language text produced. For example, given a system to be specified one can have groups writing informal contracts and other groups writing formal abstract contracts. Thereafter our tool would translate the formal abstract contracts to natural language contracts. This permits comparison between informal text produced by hand and the text produced by our tool.

Another interesting experiment would be to start from an informal specification, and then create an abstract formal specification. Then our tool would be used to translate the abstract formal specification into natural language text. Thereafter, make a comparison between the two natural language specifications.

References

1. J. R. Abrial. *B-Book*. Cambridge University Press, 1996.
2. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
3. David Burke and Kristofer Johannisson. Translating formal software specifications to natural language—a grammar-based approach. In P. Blache, E. Stabler, J. Busquets, and R. Moot, editors, *Logical Aspects of Computational Linguistics*, number 3492 in LNAI. Springer, 2005.
4. International Organisation for Standardization. Information technology—programming languages, their environments and system software interfaces, Vienna Development Method, specication language, part 1: Base language, 1996. ISO/IEC 13817-1.
5. International Organisation for Standardization. Information technology—Z formal specification notation—syntax, type system and semantics, 2000. ISO/IEC 13568:2002.
6. Martin Giese and Rogardt Heldal. From informal to formal specifications in UML. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc. of UML2004, Lisbon*, volume 3273 of *LNCS*, pages 197–211. Springer, 2004.
7. Robert L. Glass. *Software Runaways. Lessons Learned from Massive Software Project Failures*. Prentice Hall, 1998.
8. Jon Atle Gulla. A general explanation component for conceptual modeling in CASE environments. *ACM Transactions on Informal Systems*, 14(3), 1996.
9. Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in LNCS, 2002.

10. C. Larman. *Applying UML and Patterns, second edition*. Addison-Wesley, 2002.
11. B. Meyer. *Object-Oriented,Software Construction*. Prentice Hall PTR, 1997.
12. OMG. *Unified Modeling Language Specification version 1.5 (2.0)*, 2005. `http://www.omg.org/technology/documents/formal/uml.htm`.
13. Aarne Ranta. Grammatical Framework: A type-theoretical grammar formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.
14. Colette Rolland and C. Proix. A natural language approach for requirements engineering. In *Advanced Information Systems Engineering, 4th International Conference CAiSE '92*, volume 593 of *LNCS*. Springer, 1992.
15. Shane Sendall and Alfred Strohmeier. From use cases to system operation specifications. In Stuart Kent and Andy Evans, editors, *UML'2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference*, volume 1939 of *LNCS*. Springer, 2000.
16. Alfred Strohmeier, Thomas Baar, and Shane Sendall. Applying fondue to specify a drink vending machine. In *Proceedings of OCL 2.0 Workshop at UML'03*, volume 102 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.
17. Sun Microsystems. Java Card homepage, 2005. `http://java.sun.com/products/javacard/`.

Towards Using OCL for Instance-Level Queries in Domain Specific Languages

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
{dkolovos,paige,fiona}@cs.york.ac.uk

Abstract. The Object Constraint Language (OCL) provides a set of powerful facilities for navigating and querying models in the MOF metamodelling architecture. Currently, OCL queries can be expressed only in the context of MOF metamodels and UML models. This adds an additional burden to the development and use of Domain Specific Languages, which can also benefit from an instance-level querying mechanism. In an effort to address this issue, we report on ongoing work on defining a rigorous approach for aligning the OCL querying and navigation facilities, with arbitrary Domain Specific Languages to support instance-level queries. We present a case-study that demonstrates the usefulness and practicality of this approach.

1 Introduction

The MOF metamodelling architecture is a four-level integrated architecture for defining, persisting and managing modelling languages and models. At its meta-meta-model level (M3), lies the Meta Object Facility (MOF) [13], a self-defined language for building modelling languages (metamodels). At the metamodel-level (M2) exist languages defined using MOF. The most prominent example of an M2 metamodel is the Unified Modeling Language (UML) [16]. Models expressed in M2-languages are considered to belong to the model-level (M1) while instances of M1 models are placed at the instance-level (M0) (or system-level according to [11]).

The Object Constraint Language (OCL) [15] is a language originally developed to support capturing constraints in models of the MOF metamodelling architecture. However, due to its expressive and efficient model navigation and querying facilities, OCL has also been used extensively as a query language both for expressing stand-alone queries [5], and in the context of model management languages for tasks such as model transformation (e.g. QVT [14], ATL [9], YATL [17]), code generation (e.g. MOFScript [2]) and model merging (e.g. EML [6]). The navigation and querying facilities of OCL operate at two levels: at the metamodel-level (M2), it can be used to define queries in the context of the abstract syntax of a modelling language. Metamodel-level queries can then be evaluated on M1 models. Similarly, at the model-level (M1), it can be used to define queries in terms of model-specific constructs that can then be evaluated on M0 instances.

OCL is currently aligned with MOF and UML. Due to the MOF-OCL alignment, OCL queries can be expressed at the metamodel level and evaluated at the model-level

for all MOF-based languages. By contrast, instance-level queries are supported only for UML models, since OCL is not aligned with any other MOF-based languages. The reason for this is the absence, to our knowledge, of appropriate techniques in the literature and the tool-market, for aligning OCL with arbitrary DSLs to support instance-level queries. As a result, in practice, alignment needs to be implemented manually for each DSL within the context of a specific OCL execution engine. This is certainly not a trivial task, as it requires significant expertise with the internals of the engine. Moreover, even if the alignment is successfully implemented for a specific engine, the alignment specification will be highly coupled with the architecture and platform of the engine and thus hard to port or reuse in a different context. In our view, the absence of a generic high-level technique for using OCL to express instance-level queries in DSL models limits the expressive power of DSLs and consequently their usefulness as viable alternatives to UML in a practical software development environment.

To address this issue, in this paper we introduce a generic technique for aligning the OCL navigational and querying facilities with arbitrary modelling languages to support instance-level queries. The remainder of the paper is organized as follows. In Section 2 we discuss the problem of aligning OCL with arbitrary DSLs in detail and identify the key-challenges. In Section 3 we introduce our technique and discuss its rationale as well as the architecture of the infrastructure that allows us to realize it in practice. In Section 4 we provide a case study that demonstrates a working example of aligning a DSL with OCL. Finally, in Section 5 we conclude and discuss interesting issues for further research.

2 Background and Motivation

The principal difficulty in aligning OCL with arbitrary DSLs lies in the two different instantiation mechanisms used in the context of the MOF architecture, as also discussed in [10]. To illustrate this problem we discuss the two different instantiation mechanisms involved in UML 1.5. As illustrated in Figure 1, an object (e.g. : $Customer$) in a UML model is an instance of the $Object$ metaclass defined in the UML metamodel. Similarly, a class (e.g. $Customer$) is an instance of the $Class$ metaclass. Moreover, although both instances are contained in the same (M1) model, the : $Customer$ object is conceptually an instance of $Customer$ class. By convention, instances produced with that *implicit* instantiation mechanism belong to the M0 level but from a strict technical perspective, both Objects and Classes are M1 instances (instances of metaclasses defined in the M2 level). While the $M2 \rightarrow M1$ instantiation mechanism is well-defined in the MOF specification [13], there is no consensus on the semantics of the $M1 \rightarrow M0$ mechanism[12].

The presence of a loosely-defined $M1 \rightarrow M0$ instantiation mechanism renders alignment of OCL with custom DSLs to support instance-level queries particularly challenging. The reason is that an OCL engine needs to be aware of the instantiation mechanism to support built-in OCL features such as $allInstances$, $oclIsTypeOf()$ and $oclIsKindOf()$. A work-around for this problem is to use OCL expressions at the M2 level (where the instantiation mechanism is well-defined) to query M0 instances like any other M1 model elements. In this way, if we wanted to query all adult customers

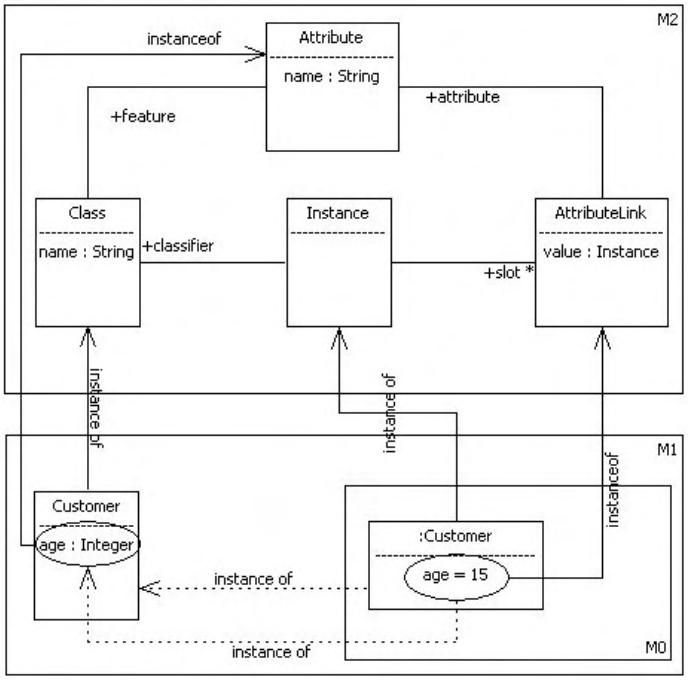


Fig. 1. Demonstration of explicit and implicit instantiation relationships in the MOF architecture

in our UML model of Figure 1, we would have to write the OCL query displayed in Listing 1.1 (or a similar one). The complexity of the OCL expression needed for such a simple query illustrates that while this approach makes querying models at the instance-level feasible, it does not scale for complex queries. By contrast, an OCL engine that is aware of the UML $M1 \rightarrow M0$ instantiation mechanism allows us to specify the same query in a much more compact and meaningful manner, as displayed in Listing 1.2.

Listing 1.1. Querying an M1-level UML model with M2-level OCL

```
1 Object.allInstances->
2   select(o :Object | o.classifier.name->includes('Customer'))->
3   select(o :Object | o.slot->exists(aL :AttributeLink |
4   aL.attribute.name = 'age' and aL.value.toInteger() > 18))
```

Listing 1.2. Quering an M1-level UML model with M1-level OCL

```
1 Customer.allInstances->select(c:Customer|c.age > 18)
```

Apart from the $M1 \rightarrow M0$ instantiation mechanism, a UML-OCL execution engine needs to be aware of the semantics of the *point* (.) navigational operator to calculate the result of expressions such as the $c.age$ in Listing 1.2. The semantics of the point operator consist of three parts; the navigation path that must be followed (in terms of M2), the multiplicity of the returned value (single value or collection) and the type of

the returned value (Integer, String, Boolean, some other user-defined type etc). Consider the M2-level query in Listing 1.1. The navigation path is defined in lines 1-4 (Object -¿ slot -¿value). The multiplicity is defined by accessing a single-valued feature (aL.value). This indicates that the result should be a single value rather than a collection. The return type is defined via explicit cast of the value of the slot to an Integer. This is done via the built-in toInteger() operation in line 4.

In summary, in order for an OCL engine to support instance-level queries for a new DSL, it must be aware of at least: the semantics of the $M1 \rightarrow M0$ instantiation mechanism and the semantics of the point navigation mechanism for the instance-level. Currently these semantics can be specified using the programming language in which the OCL engine is implemented (e.g. Java) and this is how UML-aware OCL engines, such as [3, 4, 19, 1], have been implemented so far. However, as discussed in [7], 3rd generation languages (3GL) are not particularly efficient for model navigation. Moreover, by adopting this approach, the specification of the semantics becomes bound to the proprietary architecture and platform of the OCL engine. Finally, from a technical perspective, modifying an OCL engine to support a new DSL is by far not a trivial task and this is partly justified by the fact that there is no published work, to our knowledge, on aligning an OCL engine with languages other than UML and MOF.

To address this issue in the following section we propose a generic and platform independent mechanism for specifying the required semantics: OCL itself.

3 Proposed Approach

In this section we demonstrate how we can specify the semantics of the $M1 \rightarrow M0$ instantiation mechanism and the instance-level point operators using OCL itself as the specification mechanism.

For practical reasons, in this work instead of using pure OCL we are using the Epsilon Object Language (EOL) [7], an OCL-based model management language. The reason we use EOL and not pure OCL is that from our experiments, we have found that the OCL expressions needed to specify the semantics of the instantiation mechanism and the point operator tend to be rather complex and consequently difficult to test and debug when expressed in pure OCL. This is because OCL does not support statement sequencing and therefore expressing complex queries requires deep nesting of expressions (including *if-else* expressions and variable declarations using *let* expressions) in a single statement. Instead, in EOL, complex expressions can be decomposed into sequences of simpler expressions that are both easier to read, understand and debug. However, we should stress that in principle, exactly the same functionality can be implemented in pure OCL.

3.1 Relationship between EOL and OCL

EOL supports almost all the navigational and querying facilities of OCL. However, it also supports additional features and also deviates from OCL in some aspects. Therefore, in this section we provide a brief discussion of the additional or deviant features we are using in the EOL listings that follow, for readers that are already familiar with

OCL. For a detailed discussion on EOL and its differences with OCL, readers can refer to [7].

Statement sequencing: In OCL, there is no notion of statement sequencing and, as already discussed, this can lead to extremely complex expressions that are difficult to read and debug. By contrast, in EOL statements can be separated using the semi-column (;) delimiter (similarly to Java,C++ and C#). In our view, this feature greatly enhances readability and renders it easier to debug a fragment of code.

Variable definition: The latest version of OCL (2.0) provides the let expression for creating temporary variables in the context of a single query. Similarly, EOL supports the def statement for defining variables in the context of a block of statements.

Helpers: OCL supports definition of custom operations (*helpers* according to the OCL specification) on meta-classes. Since OCL does not support statement sequencing, the body of an OCL helper is a single OCL expression. By contrast, in EOL, the body of a helper operation is a sequence of statements and values are returned using the $return$ statement.

Style: In EOL, the Ocl prefix has been removed from the names of features such as $OclAny$, $oclIsTypeOf$ or $oclIsKindOf$ (in EOL they are called Any, $isTypeOf$, $isKindOf$). Moreover, built-in operations such as $select()$ and $size()$ that are accessible using the $\rightarrow$ operation in OCL, are also accessible using the point operator in EOL.

3.2 Contents and Structure of an Alignment Specification

To align EOL with a DSL, we need to construct an *alignment specification*. Such a specification consists of the following operations (or *helpers* in OCL terms) that operate at the meta-model level and define the required semantics:

operation String allOfType() : Sequence The $allOfType$ operation applies to a String that specifies the name of the type under question and returns all the model elements that are direct instances of the type. This is needed both to be able to calculate the result of the $isTypeOf$ operation at the instance-level.

operation String allOfKind() : Sequence The $allOfKind$ operation applies on a String that specifies the name of the type under question and returns all the model elements that are either direct or indirect (through some kind of inheritance in the M1 level) instances of the type. The $allOfKind$ method is needed to be able to calculate the result of the $isKindOf$ and the $allInstances$ operations at the instance-level. The existence of both the $allOfKind$ and the $allOfType$ operations allows us to support inheritance in the model-level (if the DSL supports such a feature).

operation Type getProperty(property : String) : Any For each $Type$ of instance at the instance-level, a $getProperty$ operation must be defined that specifies the semantics of the point navigational operator in the model-level. As discussed in Section 2, a $getProperty$ operation must define: the navigation path for retrieving the value of the $property$, the multiplicity and the type of the returned value.

3.3 Implementation Architecture

A basic design principle of EOL was that it should be able to manage models of diverse metamodels and technologies. This principle is implemented in the underlying Epsilon Model Connectivity (EMC) layer. The basic concept of EMC is the $EolModel$ interface to which all EOL-compatible models must conform. Implementations of $EolModel$ include the $MdrModel$, $EmfModel$ and $XmlDocument$ that allow EOL to manage MDR [18] and EMF-based [8] models as well as XML documents. In the aforementioned implementations of $EolModel$, the required methods (e.g. $allOfType$, $allOfKind$) are specified using Java.

To align with custom DSLs we have defined a new specialization of $EolModel$ named $EolM0Model$ that delegates calls to its methods to the underlying alignment specification (instead of implementing them in Java). For example, if the instance-level query contains the $X.allInstances$ expression, the EOL engine will invoke the $allOfKind(X)$ Java-method of the $EolM0Model$ that will in its turn delegate the call to the $allOfKind(X)$ EOL operation defined in the alignment specification. This is illustrated in Figure 2.

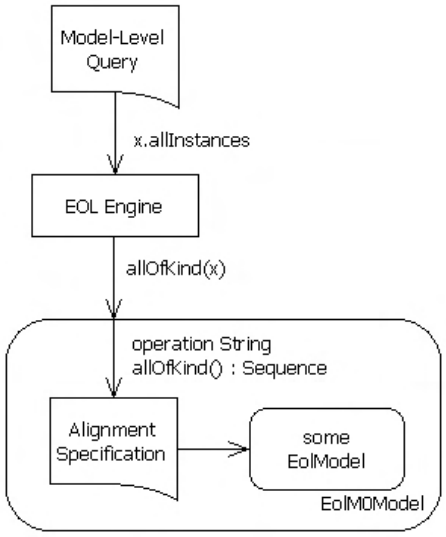


Fig. 2. Architecture of the alignment mechanism

Using this approach, to align with a new DSL, engineers do not need to be aware of the internals of the execution engine, the modelling framework (EMF, MDR etc) or write code in the implementation language of the engine (Java) at all. Instead, they need only provide a high-level alignment specification, in EOL, that implements the required operations.

4 Case Study

Having outlined our approach in Section 3, in this section we present a case-study of aligning OCL with a DSL for modelling Relational Databases. The metamodel of the Relational DSL (constructed using EMF) is presented graphically in Figure 3. There, a $Database$ consists of many tables and each $Table$ consists of a number of $Columns$. All $Database$, $Table$ and $Column$ have a $name$ and $Column$ also has a $type$. Related columns are linked each other using foreign-keys. Each $ForeignKey$ defines a $parent$ and a $child$ column and also if the relationship is one-to-one or one-to-many ($oneToMany$). In the shaded part of the metamodel the *M0* constructs[1] appear. A $TableData$ contains a set of $Rows$ that represent exemplar data of the related $table$. Finally, a Row contains many cells and each $Cell$ corresponds to a $column$ of the table and also has a $value$.

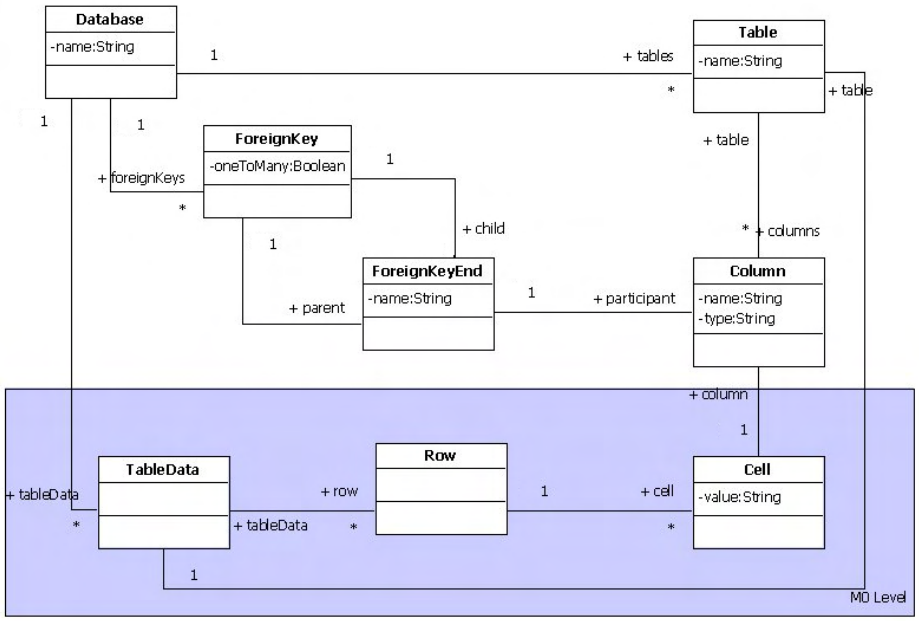


Fig. 3. The abstract syntax of a DSL for Relational Databases

[1] By *M0 constructs* of the metamodel, we refer to metamodel constructs, instances of which belong to the M0 level

A visualized version of an instance of the Relational DSL is illustrated in Figure 4. There, the two shapes on the top represent instances of $Table$ and the two shapes at the bottom represent instances of $TableData$.

4.1 Defining the $M1 \rightarrow M0$ instantiation semantics

In our DSL, a Row is conceptually an instance of a $Table$. Therefore, the expression $Customer.allInstances$ should return all the rows in the model that belong to the $TableData$ that has an associated $Table$ with the name *Customer*. This is formally defined by the $allOfKind$ operation of Listing 1.3. In Listing 1.3, the $allOfType$ operation is also defined. The fact that they both return the same result indicates that there is no notion of inheritance in our DSL.

Listing 1.3. Specification of the allOfType and allOfKind operations

```
1 operation String allOfType() : Sequence(Row) {
2       return Row.allInstances().
3         select(r|r.tableData.table.name = self);
4 }
5
6 operation String allOfKind() : Sequence(Row) {
7       return self.allOfType();
8 }
```

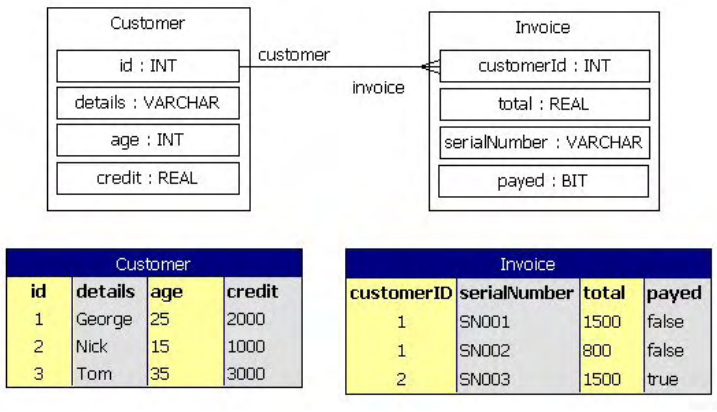


Fig. 4. An instance of the Relational DSL

4.2 Defining the point operator semantics

Having defined the $M1 \rightarrow M0$ instantiation semantics, we must now define the semantics of the point operator for the instance level. To provide better understanding,

we first describe the semantics informally through a set of small examples: Let c be the first row of the Customer table-data displayed in Figure 4. In this case, the expression $c.age$ should return an $Integer$ (25). Similarly, $c.details$ should return a $String$ (George). Moreover, $c.invoice$ should return a collection of all the rows of the Invoice table-data where the value of $customerId$ is equal to the value of $c.id$. This is dictated by the foreign key that relates the repsective columns in the model. The complete formal semantics of the point operator are captured in the $getProperty(name : String)$ operation of Listing 1.4. The $getProperty$ operation delegates the task of defining the navigation path and the multiplicity of the returned result to the $getRowsOrCell()$ operation. Finally, the $getValue()$ operation (lines 12-23), casts the string values of cells to the respective OCL data-types (Boolean, String, Integer and Real) according to the the $type$ of the respective $Column$ (BIT, VARCHAR, INT and REAL).

Listing 1.4. Specification of the getProperty operation

```
1  operation Row getProperty(name : String) : Any {
2    def ret : Any;
3    ret := self.getCellOrRows(name);
4    if (ret.isTypeOf(Cell)){
5      return ret.getValue();
6    }
7    else {
8      return ret;
9    }
10 }
11
12 operation Cell getValue() : Any {
13   if (cell.column.type = 'INT'){
14     return cell.value.asInteger();
15   }
16   if (cell.column.type = 'BIT'){
17     return cell.value.asBoolean();
18   }
19   if (cell.column.type = 'REAL'){
20     return cell.value.asReal();
21   }
22   return cell.value.asString();
23 }
24
25 operation Row getCellOrRows(name : String) : Any {
26
27   def cell : Cell;
28
29   -- First try to find a cell with that name
30   cell := self.cell.select(c|c.column.name = name).first();
31
32   if (cell.isDefined()){
33     -- If a cell with that name exists, return it
34     return cell;
35   }
```

```
36   else {
37     -- Try to find a foreign child-key with that name
38     def childKeyCell : Cell;
39
40     childKeyCell := self.cell.select
41       (c|ForeignKey.allInstances().
42         exists(fk|fk.child.participant =
43           c.column and fk.parent.name = name)).first();
44
45     if (childKeyCell.isDefined()) {
46       def ck : ForeignKey;
47       ck := ForeignKey.allInstances().
48         select(fk|fk.child.participant = childKeyCell.column)
49           .first();
50       return Row.allInstances().
51         select(r|r.cell.exists(c|c.column = ck.parent.participant
52           and c.value = childKeyCell.value)).first();
53     }
54     else {
55       -- Try to find a foreign parent-key with that name
56       def parentKeyCell : Cell;
57       parentKeyCell := self.cell.select
58         (c|ForeignKey.allInstances()
59           .exists(fk|fk.parent.participant = c.column
60             and fk.child.name = name)).first();
61
62       if (parentKeyCell.isDefined()) {
63         def pk : ForeignKey;
64         pk := ForeignKey.allInstances().
65           select(fk|fk.parent.participant = parentKeyCell.column)
66             .first();
67         def rows : Sequence(Row);
68         rows := Row.allInstances().
69           select(r|r.cell.exists(c|c.column = pk.child.participant and
70             c.value=parentKeyCell.value));
71         if (pk.oneToMany){
72           return rows;
73         }
74         else{
75           return rows.first();
76         }
77       }
78
79     }
80
81   }
82   throw 'Undefined property: ' + name;
83 }
```

4.3 Running instance-level queries on the model

Having defined the alignment specification, we can now express and evaluate OCL instance-level queries on our model. The OCL expression of Listing 1.5 returns a $Collection$ of the $details$ of all the customers in our model that have an age under 18 ('Nick'). In a more complex query, Listing 1.6 prints a message for every customer that has unpayed invoices, the sum of which exceed his/her credit.

Listing 1.5. Instance-level query for retrieving under-aged customers

```
1 Customer.allInstances.select(c|c.age < 18).collect(c|c.details);
```

Listing 1.6. Instance-level query for retrieving customers in debt

```
1  for (c in Customer.allInstances){
2
3    def balance : Real;
4
5    balance := c.invoice.select(i|i.payed = false)
6      .collect(i|i.total).sum();
7
8    if (balance > c.credit){
9      ('Customer ' + c.details + ' has a negative balance').println();
10   }
11 }
```

5 Conclusions and Further Work

In this paper we have presented a novel technique for aligning OCL with custom Domain Specific Languages to support instance-level queries. Moreover, we have presented a working example of applying this technique in a DSL for modelling Relational Databases that demonstrates its practicality and usefulness. However, we plan to continue our experiments with diverse metamodels to further validate or refine (where this is required) our approach.

As discussed in Section 3, we are using EOL instead of pure OCL for defining the alignment specification. This is primarily due to the practical difficulties involved in capturing complex expressions such as this displayed in the $getRowOrCells$ operation of Listing 1.4 using pure OCL. However, we realize that expressing the alignment specification in that way renders re-use from plain OCL engines impossible. Therefore, we are considering developing a transformation from EOL to pure OCL that will be able to translate sequential EOL statements into a single OCL-compatible statement.

6 Acknowledgements

The work in this paper was supported by the European Commission via the MODELWARE project. The MODELWARE project is co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006).

References

1. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net.
2. MOFScript. Official Web-Site: http://www.modelbased.net/mofscript/.
3. OCLE: Object Constraint Language Environment, official web-site. http://lci.cs.ubbcluj.ro/ocle/.
4. Octopus: OCL Tool for Precise Uml Specifications, official web-site. http://www.klasse.nl/ocl/octopus-intro.html.
5. D. H. Akehurst and Behzad Bordbar. On Querying UML Data Models with OCL. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 91–103, London, UK, 2001. Springer-Verlag.
6. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, Genova, Italy, October 2006. LNCS. (to appear).
7. Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.
8. Eclipse.org. Eclipse Modelling Framework. http://www.eclipse.org/emf.
9. Frédéric Jouault and Ivan Kurtev. Transforming Models with the ATL. In Jean-Michel Bruel, editor, *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2005.
10. Ivan Kurtev, Klaas van den Berg. Unifying Approach for Model Transformations in the MOF Metamodeling Architecture. Technical Report TR-CTIT-04-12, University of Twente, 2004. CTIT Technical Report, ISSN 1381-3625.
11. Jean Bezivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.
12. Jean Bézivin and Richard Lemesle. Ontology-Based Layered Semantics for Precise OA&D Modeling. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 151–154, London, UK, 1998. Springer-Verlag.
13. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
14. Object Management Group. MOF QVT Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf.
15. Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf.
16. Object Management Group. UML official web-site. http://www.uml.org.
17. Octavian Patrascoiu and Peter Rodgers. Embedding OCL Expressions in YATL. In *Proc. OCL and Model Driven Engineering wokshop, UML'04*, October 2004.
18. Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.
19. University of Bremen, Database Systems Group. USE - A UML-based Specification Environment. http://www.db.informatik.uni-bremen.de/projects/USE/.

OCL-based Validation of a Railway Domain Profile

Kirsten Berkenkötter

University of Bremen,
P.O. Box 330 440
28334 Bremen, Germany
kirsten@informatik.uni-bremen.de

Abstract. Domain-specific languages become more and more important these days as they facilitate the close collaboration of domain experts and software developers. One effect of this general tendency is the increasing number of UML profiles. UML itself as the most popular modeling language is capable of modeling all kinds of systems but it is often inefficient due to its wide-spectrum approach. Profiles tailor the UML to a specific domain and can hence be seen as domain-specific dialects of UML. At the moment, profiles mainly introduce new terminology, often in combination with OCL constraints which describe the new constructs more precisely. As most tools do not support validation of OCL expressions let alone supplementing profiles with OCL constraints, it is difficult to check if models based on a profile comply to this profile. A related problem is checking whether constraints in the profile contradict constraints in the UML specification. In this paper, it is shown how to complete these tasks with the tool USE. As an example, a profile from the railway control systems domain is taken which describes the use of its modeling elements quite strictly. Models based on this profile serve as a foundation for automated code generation. Therefore, they require a rigorous and unambiguous meaning. OCL is heavily used to reach this goal.

1 Introduction

The current interest in model driven architecture (MDA) [OMG03] and its surrounding techniques like metamodeling and model driven development (MDD) has also increased the interest in domain-specific languages (DSL) and their development. MDA enforces the idea of platform independent models (PIM) as main artifact in the design of software systems, while the concrete implementation will be based on a platform specific model (PSM). The step from PIM to PSM is performed by transformations while the generation of code is based on the PSM and a description of the concrete target platform called platform model (PM).

In the context of MDA, several standards have been developed like the Meta Object Facility (MOF) [OMG06] for designing metamodels and the Unified Modeling Language (UML) [OMG05c,OMG05b] as a modeling language. UML has

become the de-facto standard for modeling languages and is supported by various tools. Due to its wide-spectrum approach, it can be used for modeling all kinds of systems. This is an advantage as one tool can be used to develop different kinds of systems. In contrast, it may also lead to inefficiency and inaccuracy as each domain has its own need, e.g. domain-specific terminology that differs from the one of UML may lead to misunderstandings. Another problem are semantic variation points in UML. These are necessary to enable the wide-spectrum approach but not useful if the model is to be used in the MDA context as transformations and code generation cannot be utilized with an ambiguous model as foundation.

A good example are railway control systems that are described in specific terminology and notation. The domain of control are track networks that consist of elements like segments, points, or signals. Routes are defined to describe how trains travel on the network. In addition, there are rules that specify in which way a network is constructed and how it is operated. Some rules apply to all kinds of railway systems and some are specific for each kind of railway system, e.g. tramway or railroads. In principle, UML is capable of modeling such systems: class diagrams can be used to describe segments, points, and other track elements and their dependencies while object diagrams model concrete track layouts and routes. Rules can be specified by means of OCL. The problem is that we have to model each kind of railway system with all rules explicitly. The domain knowledge that covers the common parts of all railway control systems is not captured in such models. Neither is specific notation that is used in the domain like symbols for signals and sensors.

Domain-specific languages are a means to overcome these disadvantages [Eva06]. Designing a new modeling language from scratch is obviously time-consuming and costly, therefore UML profiles have become a popular mechanism to tailor the UML to specific domains. In this way, different UML dialects have been developed with considerably low effort. New terminology based on existing UML constructs is introduced and further supplied with OCL [OMG05a,WK04] constraints to specify its usage precisely. Semantics are often described in natural language just as for UML itself.

With respect to railways, the Railway Control Systems Domain (RCSD) profile has been developed [BHP,BH06] as domain-specific UML derivative with formal semantics.The main reason for developing this profile was to simplify the collaboration of domain experts of the railway domain and software developers that design controllers for this domain. With the help of the profile, the system expert develops track networks for different kind of railway systems consisting of track segment, signal, points, etc. The software specialist works on the same information to develop controllers. In the end, controller code which satisfies safety-critical requirements shall be generated automatically. Railway control systems are especially interesting as the domain knowledge gathered in the long history of the domain has to be preserved while combining it with development techniques for safety-critical systems. Structural aspects are specified by class and object diagrams (see Fig. 8 and Fig. 9) whose compliance to the domain

is ensured by OCL constraints. Semantics are based on a timed state transition system that serves as foundation for formal transformations towards code generation for controllers as well as for verification tasks. In this paper, the focus is on the validation of the structural aspects to ensure the correct and successful application of transformations and verification. Details about semantics can be found in [PBD+05,BH06,BHP].

A problem that has not been tackled until now is to validate that the constraints of a profile comply to the ones of UML and that models using a profile comply to this profile. One reason for this is that CASE tools often support profiles as far as new terminology can be introduced but lack support of OCL [BCC+05]. One of the few tools that support OCL is USE (UML Specification Environment) [Ric02,GZ04]. It allows the definition of a metamodel supplied with OCL constraints and checks whether models based on this metamodel fulfill all constraints. Using (a part of) the UML metamodel in combination with a profile as the USE metamodel allows for fulfilling three goals: (a) Validating that this profile complies to the UML metamodel as each model has to fulfill the invariants of the UML metamodel and the profile. (b) Validating that class diagrams comply to the profile. (c) Validating that object diagrams comply to the profile if the profile describes instances as well as instantiable elements. This approach has been used to validate the RCSD profile and models based on this profile.

The paper is organized in the following way: the next section gives an introduction to UML profiles and the usage of OCL in this context. After that, the railway domain is briefly introduced in Sec. 3, followed by a description of the RCSD profile and typical constraints in Sec. 4. After that, Sec. 5 describes the validation with USE on the different levels. At last, the results of this validation approach and future work are discussed in Sec. 6.

2 UML Profiles and OCL

UML profiles as described in in [OMG05b] and [OMG05c] offer the possibility to tailor the UML to a specific domain in several ways: (a) introducing new terminology, (b) introducing new syntax/notation, (c) introducing new constraints, (d) introducing new semantics, and (e) introducing further information like transformation rules.

Changing the existing metamodel itself e.g. by introducing semantics contrary to the existing ones or removing elements is not allowed. Consequently, each model that uses profiles is a valid UML model. Profiles are therefore not a means to develop domain-specific languages that contradict UML constraints or semantics. Due to the wide-spectrum approach of UML, semantics are loosely enough to allow all kinds of profiles. A UML 2.0 profile mainly consists of stereotypes, i.e. extensions of already existing UML modeling elements. You have to choose which element should be extended and define the add-ons. In addition, new primitive datatypes and enumerations can be defined as necessary.

OCL can be used in various ways to specify the stereotypes more precisely:

(a) Constraining property values: A stereotype has all properties of its base class and can add only attributes. Defining new associations to classes in the reference metamodel or other stereotypes is not allowed. Therefore, constraining values of existing attributes and associations is a useful means to give a stereotype the desired functionality.
(b) Specifying dependencies between values of different properties of one element: Often, it is necessary to describe dependencies between the properties of a modeling element precisely.
(c) Specifying dependencies between property values of different instances of one element: Some properties like identification numbers need specific values for different instances of one element.
(d) Specifying dependencies between property values of different instances of different elements: In the same way, several elements may have properties whose values have some kind of relationship. Here, it is important to chose the context of the constraint carefully such that the constraint is not unnecessarily complicated because another modeling element would have been the better choice as basis for the constraint.

3 Short Introduction to the Railway Domain

Creating a domain specific profile requires identifying the elements of this domain and their properties as e.g. described in [Pac02]. In the railway domain, track elements, sensors, signals, automatic train runnings, and routes have been proven essential modeling elements. They are described shortly in the following, more details can be found in [BH06]:

Track Elements The track network consists of segments, crossings, and points. Segments are rails with two ends, while crossings consist of either two crossing segments or two interlaced segments. Points allow a changeover from one segment to another one. Single points have a stem and a branch. Single slip points and double slip points are crossings with one, respectively two, changeover possibilities.

Sensors Sensors are used to identify the position of trains on the track network, i.e. the current track element. To achieve this goal, track elements have entry and exit sensors located at each end. The number of sensors depends on the allowed driving directions, i.e. the uni- or bidirectional usage of the track element.

Signals Signals come in various ways. In general, they indicate if a train may go or if it has to stop. The permission to go may be constrained, e.g. by speed limits or by obligatory directions in case of points. As it is significant to know if a train moves according to signaling, signals are always located at sensors.

Automatic Train Running Automatic train running systems are used to enforce braking of trains, usually in safety-critical situations. The brake enforcement may be permanent or controlled, i.e. it can be switched on and off. Automatic train running systems are also located at sensors.

Route Definition As sensors are used as connection between track elements, routes of a track network are defined by sequences of sensors. They can be

entered if the required signal setting of the first signal of the route is set. This can only be done if all points are in the correct position needed for this route. Conflicting routes cannot be released at the same time.

4 RCSD Profile

Unfortunately, defining eight stereotypes as suggested by the domain analysis in Sec. 3 is not sufficient. New primitive datatypes, enumerations, and special kinds of association to model interrelationships between stereotypes are needed. Furthermore, UML supports two modeling layers, i.e. the model layer itself (class diagrams) and the instances layer (object diagrams). In the RCSD profile, both layers are needed: class diagrams are used to model specific parts of the railway domain, e.g. tramways or railroad models, while object diagrams show explicit track layouts for such models. Hence, stereotypes on the object level have to be defined. For these reasons, the RCSD profile is structured in five parts: the definition of primitive datatypes and literals, network elements on class level, associations between these elements, instances of network elements and associations, and route definitions.

4.1 Types and Literals

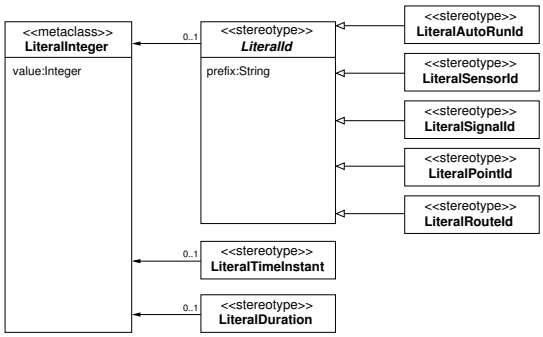


Fig. 1. Literals part of the RCSD profile

Several new datatypes are needed: identifiers for all controllable elements, identifiers for routes (e.g. to specify conflicting ones), time instants and durations. All of them have in common that the value domain is $\mathbb{N}$. Defining different datatypes facilitates constraints like: all signal identifiers are unique, all point identifiers are unique and so on. In addition, each new datatype has a dedicated stereotype to model literals of this type (see Fig. 1). For the identification types, the corresponding literal consists of an integer value and a prefix character. Literals for time instants and durations are integer values.

```
inv LiteralPointId1:
  value >= 0
inv LiteralPointId2:
  prefix = 'P'
```

OCL constraints for these stereotypes are simple as only values of properties are restricted. Integers values have to be from $\mathbb{N}$; prefixes for different identification types have specific values: *'S'* for sensors, *'Sig'* for signals, *'P'* for points, *'A'* for automatic runnings, and *'R'* for routes. As an example, the two constraints needed for *LiteralPointId* are given above. For the sake of brevity, the name of invariants and the invariants context, where it is unmistakable, are omitted in the following.

4.2 Network Elements

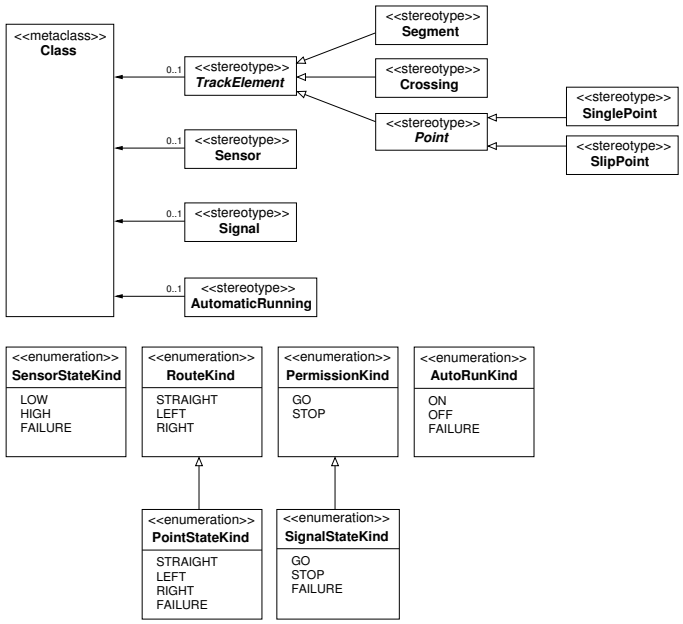


Fig. 2. Network elements of the RCSD profile

The next part of the profile defines track network elements, i.e. segments, crossing, points, signals, sensors, and automatic train runnings (see Fig. 2). *Segment*, *Crossing*, and *Point* have in common that they form the track network itself, therefore they are all subclasses of the abstract *TrackElement*. Similarly, *SinglePoint* and *SlipPoint* are specializations of *Point*. Enumerations are defined to specify values of properties. All elements are equipped with a set of constraints that define which properties must be supported by each element and how it is related to other elements.

An instance of *TrackElement* on the model layer must provide several properties: *maximalNumberofTrains* to restrict the number of trains on a track element at one point in time (mandatory) and *limit* to give a speed limit (optional). Both properties have to be integers. The first one has a fixed multiplicity 1, the second one may have multiplicities 0..1 or 1. Such requirements for *TrackElement* are defined in the following way:

```
ownedAttribute->one(a | a.name->includes('maxNumberOfTrains') and
                        a.type.name->includes('Integer') and
                        a.upperBound() = 1 and a.lowerBound() = 1 and
                        a.isReadOnly = true)
```

At each end of a *TrackElement*, entry or exit sensors can be associated. *e1Entry*, *e1Exit*, *e2Entry*, and *e2Exit* are used to model these ends of associations to sensors (optional). All outgoing associations must be *SensorAssociations*.

```
ownedAttribute->one(a | a.name->includes('e1Entry') and
                        a.upperBound() = 1 and a.lowerBound() >= 0 and
                        a.isReadOnly = true and
                        a.outgoingAssociation.
                          oclIsTypeOf(SensorAssociation)) or
(not ownedAttribute->exists(a2 | a2.name->includes('e1Entry')))
...
ownedAttribute->collect(outgoingAssociation)->
    forAll(a | a.oclIsTypeOf(SensorAssociation) or a.isUndefined)
```

To understand the structure of these constraints, a look at the UML metamodel is helpful. As all network elements are stereotypes of *Class* from the UML 2.0 Kernel package, we can refer to all properties of *Class* in our constraints. Properties on the model level are instances of class *Property* on the metamodel level, which are associated to *Class* by *ownedAttribute*. As a *StructuralFeature*, *Property* is also a *NamedElement*, a *TypedElement*, and a *MultiplicityElement*, which allows to restrain name, type, and multiplicity as shown in the constraints above. Such constraints are defined for all network elements. They all belong obviously to the category (a) as described in Sec. 2. They restrict properties on the metamodel level for the usage on the model level.

4.3 Associations

Three types of associations are defined: *SensorAssociation* that connect track elements and sensors, *SignalAssociations* that connect signals and sensors, and *AutoRunAssociations* that connect automatic train runnings and sensors (see Fig. 5). Constraints are needed e.g. to determine the kind of stereotype at the ends of each association and their number. As an example, each *SignalAssociation* is connected to one sensor and one signal:

```
inv SignalAssociation1: memberEnd->size() = 2
inv SignalAssociation2: endType->size() = 2
inv SignalAssociation3: endType->one(t | t.oclIsKindOf(Sensor))
inv SignalAssociation4: endType->one(t | t.oclIsKindOf(Signal))
```

Similar constraints are defined for the other kinds of association.

4.4 Instances of Network Elements and Associations

For each non-abstract modeling element and each association, there exists a corresponding instance stereotype (see Fig. 6). Here, the domain-specific notation is defined. In Fig. 3, two unidirectional segments connected by a sensor *S1* are shown. For comparison, the same constellation in object notation is given in Fig. 4.

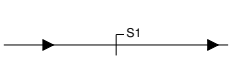


Fig. 3. RCSD notation

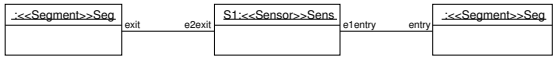


Fig. 4. UML notation

The instances are heavily restricted by OCL constraints as the instance level serves as the basis for automated code generation. Again, we find several constraints of category (a), where the values of properties are specified explicitly. To give an example, the maximal number of trains on a crossing or point is always defined and the value is 1:

```
slot->one(s1 | s1.definingFeature.name->includes('maxNumberOfTrains') and
               s1.value->size()= 1 and
               s1.value->first().oclIsTypeOf(LiteralInteger) and
               s1.value->first()->oclAsType(LiteralInteger).value = 1)
```

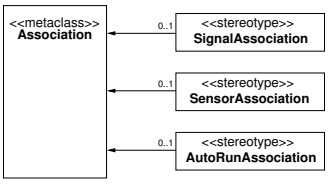


Fig. 5. Associations part of the RCSD profile

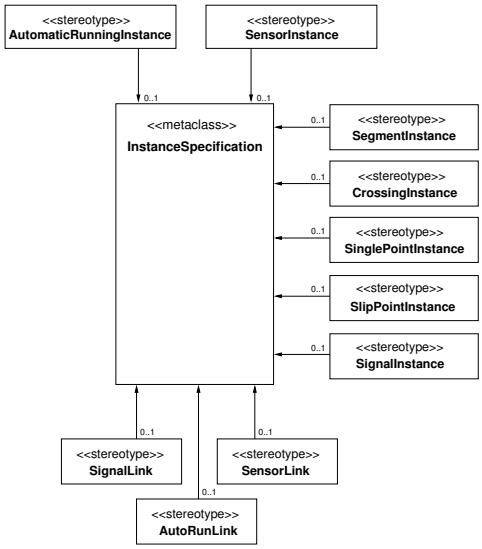


Fig. 6. Instances of network elements and associations part of the RCSD profile

Similar constraints appear for all kinds of track elements, e.g. the limit on track elements must have a value from $\mathbb{N}$ if present. More interesting are the constraints from category (b) that describe the dependencies between properties of one

stereotype. As an example, each *Point* has a *plus* and *minus* position. One of these has to be *STRAIGHT* and the other one *LEFT* or *RIGHT*:

```
slot->select(s1 | s1.definingFeature.name->includes('minus') or
                  s1.definingFeature.name->includes('plus'))->
  one(s2 | s2.value->size()= 1 and
           s2.value->first().oclIsTypeOf(InstanceValue) and
           s2.value->first().oclAsType(InstanceValue).instance.name->
             includes('STRAIGHT')) and
slot->select(s1 | s1.definingFeature.name->includes('minus') or
                  s1.definingFeature.name->includes('plus'))->
  one(s2 | s2.value->size()= 1 and
           s2.value->first().oclIsTypeOf(InstanceValue) and
           (s2.value->first().oclAsType(InstanceValue).instance.name->
              includes('LEFT') or
            s2.value->first()->oclAsType(InstanceValue).instance.name->
              includes('RIGHT')))
```

An example from category (c) are identification numbers of sensors that have to be unique. Each *Sensor* must have a property *sensorId* that is unique with respect to all instances of *Sensor*:

```
SensorInstance.allInstances->collect(slot)->asSet->flatten->
  select(s | s.definingFeature.name->includes('sensorId'))->
    iterate(s:Slot;
            result:Set(LiteralSensorId) = oclEmpty(Set(LiteralSensorId)) |
            result->including(s.value->first.oclAsType(LiteralSensorId)))->
              isUnique(value)
```

4.5 Route definitions

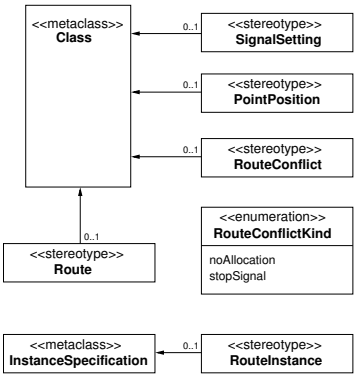


Fig. 7. Route definition part of the RCSD profile

Moreover, the profile defines routes and their instances as shown in Fig. 7. Each *Route* is defined by an ordered sequence of sensors. The signal setting for entering the route and sets of required point positions and of conflicts with

other routes are further necessary information. Again, constraints are used for unambiguous and strict definitions of properties. Constraints from category (d) are typical as sensors, signals, and points are referenced by their id in route definitions. This implies that these ids belong to some existing instances, e.g. the sensor ids given in the definition of a route. Hence, the following constraint must hold for each *RouteInstance*:

```
let i:Set(Integer) =
  slot->select(s | s.definingFeature.name->includes('routeDefinition'))->
    asSequence->first().value->
      iterate(v:ValueSpecification;
              result:Set(Integer)=oclEmpty(Set(Integer)) |
              result->including(v.oclAsType(LiteralSensorId).value))
in
  i->forAll(id | SensorInstance.allInstances->exists(sens |
    sens.slot->select(s | s.definingFeature.name->includes('sensorId'))->
      asSequence->first().value->first().
        oclAsType(LiteralSensorId).value = id))
```

5 Validation of Wellformedness Rules with USE

The next step is adapting the profile and its various invariants to USE for the validation process. USE expects a model in textual notation as input. For syntax details, we refer to [GZ04]. In our case, this is the metamodel consisting of (a part of) the UML metamodel and the profile. On this basis, instance models can be checked with respect to the invariants in the metamodel. In our case, the instance model consists of both class layer and object layer, i.e. models using the RCSD profile. A similar application of USE with respect to the four metamodeling layers of UML is shown in [GFB05].

This metamodel file includes both the necessary part of the UML 2.0 metamodel and the RCSD profile for two reasons: first, the profile cannot exist without its reference metamodel and second, one goal is to check the compliance of the profile to the metamodel. This task must be performed implicitly as USE does not check if the given constraints contradict. Instead, we assume the profile compliant to the metamodel as long as both the constraints in the metamodel and the constraints in the profile are all valid. Contradicting constraints can be identified if all constraints in the profile evaluate to true but some constraint(s) in the metamodel evaluate(s) to false.

5.1 Modeling the UML Metamodel and the RCSD Profile for USE

In the metamodel file, a description of classes with attributes and operations, associations, and OCL constraints is expected. OCL constraints are either invariants as shown in Sec.4, definitions of operations, or pre-and postconditions of operations. Only operations whose return value is directly specified in OCL and not dependent on preconditions are considered side-effect free and may be used

in invariants. For the validation of the profile, all invariants must be fulfilled by the instance model(s).

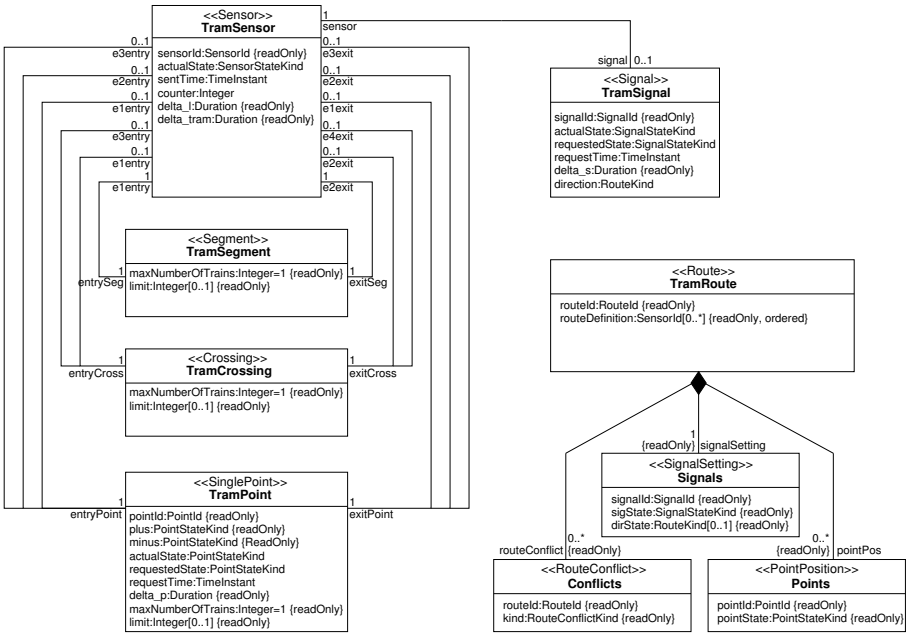


Fig. 8. Tram network definitions - class level

From the UML metamodel, the *Kernel* package has been modeled with some modifications: (a) Packages are not needed by the RCSD profile and therefore skipped in all diagrams, diagram *Packages* has been omitted completely. (b) Lower and upper bounds of multiplicities have been changed to *LiteralInteger* instead of *ValueSpecification* for easier handling. One reason is that the invariants in the context of *MultiplicityElement* are not specific enough to guarantee that the *ValueSpecification* really evaluates to *LiteralInteger* as necessary. Therefore, expressions cannot be used to specify multiplicities. The invariants of *MultiplicityElement* have been adapted to this. (c) Several invariants and operations had to be rewritten or omitted completely as they are erroneous in the UML specification. More information about this problem can be found in [BGG04]. (d) Some names in the UML specification had to be changed due to conflicts with USE keywords or multiple usage in the specification which also leads to conflicts. This problem is also described in [BGG04]. (e) USE does not support *UnlimitedNatural* as type. This problem has been overcome by using *Integer* and additional constraints that restrict corresponding values to $\mathbb{N}$. All in all, 34 invariants have been modeled here. Further packages from the UML metamodel are not needed.

Profiles are not directly supported by USE. This problem has been overcome by modeling each stereotype as a subclass from its metaclass, i.e. a metamodel extension. Modeling profiles as restricted extensions to metamodels is feasible with respect to [JSZ+04]. Here, modifications to metamodels are classified in

level one (all extensions to the reference metamodel allowed), level two (new constructs can be added to the referenced metamodel, but existing ones cannot be changed), level three (each new construct must have a parent in the reference metamodel), and level four (new relationships are only allowed as far as existing ones are specialized. The lower levels include all restrictions of the levels above. Therefore, profiles can be considered a level four metamodel extension and modeled as such in USE. [1] All in all, the following invariants of types (a) - (d) have been specified:

Profile part	(a)	(b)	(c)	(d)
Types and Literals	12	0	0	0
Network Elements	92	0	0	0
Associations	23	0	0	0
Instances	101	21	5	0
Route Definitions	36	4	1	16

5.2 Compliance of RCSD Model to Profile on Class Level

Evaluating constraints is possible for instances of the given (meta)model. As an example, a tram network description is used on class level. Tram networks consist of segments, crossing and single points that are all used unidirectionally. Furthermore, there are signals, sensors, and routes, but no automatic runnings. This constellation is shown in Fig. 8.

In USE, an instance model can be constructed step by step by adding instances of classes and associations of the metamodel to an instance diagram. More convenient is the usage of a **.cmd* command file where instance creation and setting of property values are specified in textual notation. Again, we refer to [GZ04] for syntax details.

5.3 Compliance of RCSD Model to Profile on Instance Level

A concrete network of a tram maintenance site with six routes is shown in Fig. 9. Note that this is diagram is given in RCSD notation and can also be shown in UML object notation as discussed in Sec. 4. The explicit route definitions have been omitted for the sake of brevity, but can be easily extracted from Fig. 9. This diagram has been used for the validation on the instance level. It consists of 12 segments, 3 crossing, 6 points, 25 sensors, 3 signals, and 6 routes, specified in a second **.cmd* file. The two **.cmd* files form a complete instance model of the metamodel consisting of classes and their instances.

5.4 Results

In this example, all invariants have been fulfilled. The correctness of the OCL constraints could be easily checked by adding intentional errors like incorrect association ends or signals with the same id. USE facilitates tracing of such

[1] [JSZ+04] considers profiles as level three which is incorrect as the relationship restriction has to be respected by profiles.

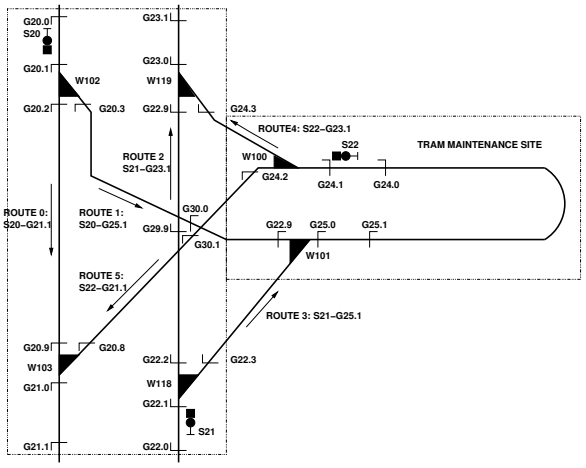


Fig. 9. Concrete track network - instance level

errors by (a) showing which instance of the metamodel has violated an invariant and by (b) decomposing the invariant in all sub-clauses and giving the respective evaluation. In Fig. 10, we can see that *sensor2* and *sensor3* have a duplicate identification numbers.

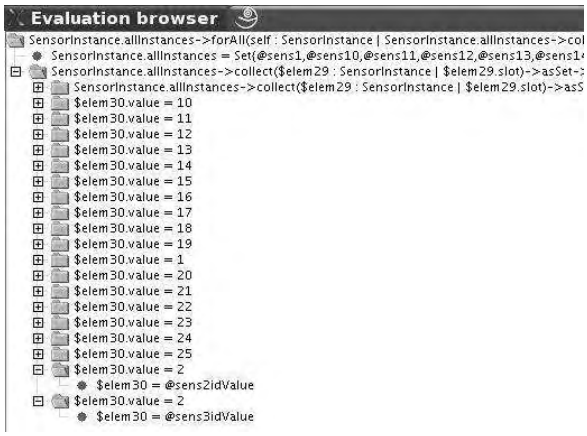


Fig. 10. Evaluation example - two identical sensor ids

For the validation process, some effort has to be made for the modeling part. Fortunately, the metamodel and profile have to be modeled only once for each profile. The part of the UML metamodel that has to be included varies from profile to profile depending on the metaclasses references by stereotypes. The current version of the USE model file consists of approximately 4000 lines. As this task is performed once per profile, the effort seems reasonable. With respect of the RCSD profile, the instance model on class level has to be modeled once per

specific railway system, e.g. once for trams. With this part of the instance model, all kinds of concrete track layouts can be checked. The tram example consists of approximately 1500 lines of input data to USE. These can be generated from class diagrams by parsing the output of CASE tools and adapting them to USE. Concrete track layout can also be generated, this time from object diagrams. In this way, all kinds of track layouts for one system can be checked. The example track layout needs about 5000 lines. As writing them for each layout would be an obnoxious task, automation is highly required.

6 Conclusion

The validation of models of the RCSD profile and the profile itself based on OCL constraints with USE has been proven useful in several ways. It has been shown that the profile complies to UML as it is required and that an example model for tramways is valid in the RCSD context. This makes object diagrams for such tramways applicable for transformation and verification purposes. Another effect of the validation with USE was the improvement of the OCL constraints themselves. As most case tools have no OCL support, it is hard to detect if constraints exhibit syntax errors or if complicated constraints really have the intended meaning.

An adaption of the validation process to other profiles can be performed straightforward as the same kinds of constraints should appear. It is possible that the UML metamodel part has to be enhanced for other profiles as this depends on the metaclasses referenced by stereotypes. Validation is reasonable in each profile whose application relies on a solid and unambiguous model.

With respect to the RCSD profile, future work has to investigate the behavioral aspects of track layouts as described in [BH06]. At the moment, only statical aspects have been examined, but USE can also be applied to the validation and test of controllers that have been generated for a concrete track network. Passing trains have to be simulated by changes of sensor values just as route requests by trains to the controller. Signals and points have to be switched by the controller with respect to safety conditions like 'only one tram on a point at one point in time' or 'only one tram on conflicting routes'. Such safety requirements can also be expressed in OCL. As train movements and signal and point switches are all modeled by variables in the track network, the outcome is always an object diagram with changed variable values whose invariants can be checked.

Acknowledgments Special thanks go to Fabian Büttner and Arne Lindow for their help with USE and to Ulrich Hannemann for his valuable feedback to the first versions of this paper and the related work.

References

[BCC+05] Thomas Baar, Dan Chiorean, Alexandre Correa, Martin Gogolla, Heinrich Hußmann, Octavian Patrascoiu, Peter H. Schmitt, and Jos Warmer. Tool

Support for OCL and Related Formalisms - Needs and Trends. In Jean-Michel Bruel, editor, *Satellite Events at the ModELS'2005 Conference*, volume 3844 of *LNCS*, pages 1–9. Springer-Verlag, 2005.

[BGG04] Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche. Detecting OCL Traps in the UML 2.0 Superstructure. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proceedings 7th International Conference Unified Modeling Language (UML'2004)*, volume 3273 of *LNCS*, pages 188–197. Springer-Verlag, 2004.

[BH06] Kirsten Berkenkötter and Ulrich Hannemann. Modeling the railway control domain rigorously with a uml 2.0 profile. In J. Górski, editor, *Safecomp 2006*, volume 4166 of *LNCS*, pages 398–411. Springer, 2006. to appear.

[BHP] Kirsten Berkenkötter, Ulrich Hannemann, and Jan Peleska. The railway control system domain. Draft, http://www.informatik.uni-bremen.de/agbs/research/RCSD/.

[Eva06] Andy Evans. Domain Specific Languages and MDA. http://www.xactium.com, 2006.

[GFB05] Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. Technical Report LGL-REPORT-2005-001, Ecole Polytechnique Fédérale de Lausanne, 2005.

[GZ04] Martin Gogolla and Paul Ziemann. Checking BART Test Scenarios with UML's Object Constraint Language. Formal Methods for Embedded Distributed Systems - How to master the complexity. Fabrice Kordon, Michel Lemoine (Eds.), Kluwer, Boston. pages 133-170, 2004.

[JSZ+04] Yanbing Jiang, Weizhong Shao, Lu Zhang, Zhiyi Ma, Xiangwen Meng, and Haohai Ma. On the Classification of UML's Meta Model Extension Mechanism. In *The Unified Modelling Language: Modelling Languages and Applications*, pages 54–68, 2004.

[OMG03] Object Management Group. MDA Guide Version 1.0.1, June 2003.

[OMG05a] Object Management Group. OCL 2.0 Specification, version 2.0. http://www.omg.org/docs/ptc/05-06-06.pdf, June 2005.

[OMG05b] Object Management Group. Unified Modeling Language: Superstructure, version 2.0. http://www.omg.org/docs/formal/05-07-04.pdf, July 2005.

[OMG05c] Object Management Group. Unified Modeling Language (UML) Specification: Infrastructure, version 2.0. http://www.omg.org/docs/ptc/04-10-14.pdf, July 2005.

[OMG06] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/docs/formal/06-01-01.pdf, January 2006.

[Pac02] Joern Pachl. *Railway Operation and Control*. VTD Rail Publishing, Mountlake Terrace (USA), 2002. ISBN 0-9719915-1-0.

[PBD+05] Jan Peleska, Kirsten Berkenkötter, Rolf Drechsler, Daniel Große, Ulrich Hannemann, Anne E. Haxthausen, and Sebastian Kinder. Domain-specific formalisms and model-driven development for railway control systems. In *TRain workshop at SEFM2005*, September 2005.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*, volume 14 of *BISS Monographs*. Logos Verlag, Berlin, 2002. Ph.D. thesis, Universität Bremen.

[WK04] Jos Warmer and Anneke Kleppe. *Object Constraint Language 2.0*. MITP-Verlag, Bonn, 2004.

Use of OCL in a Model Assessment Framework: An experience report

Joanna Chimiak–Opoka, Chris Lenz

Quality Engineering Research Group
Institute of Computer Science, University of Innsbruck
Technikerstrasse 21a, A–6020 Innsbruck
joanna.opoka@uibk.ac.at

Abstract. In a model assessment framework different quality aspects can be examined. In our approach we consider consistency and perceived semantic quality. The former can be supported by constraints and the later by queries. Consistency can be checked automatically, while for the semantic quality the human judgement is necessary. For constraint and query definitions the utilisation of a query language was necessary. We present a case study that evaluates the expressiveness of the Object Constraint Language (OCL) in the context of our approach. We focus on typical queries required by our methodology and we showed how they can be formulated in OCL. To take full advantage of the language's expressiveness, we utilise new features of OCL 2.0. Based on our examination we decided to use OCL in our analysis tool and we designed an architecture based on Eclipse Modeling Framework Technology.

1 Introduction

The necessity of model maintenance is growing together with the increasing size of models used in real applications. The importance of **integration** grows with the size and the number of designed models. The aspect of integration becomes crucial if the modelling environment is not homogeneous, i.e., it has to be dealt with diverse modelling tools and even with diverse notations. Such a situation is common if various aspects of the same system have to be described. For example in the domain of enterprise architecture modelling, for the description of business processes and technical infrastructure different tools and notations can be used.

If additionally the models are large scale models with hundreds or thousands of elements they might very likely contain inconsistencies and gaps. Quality assurance of these models can not be done by pure manual inspection or review but requires tool assistance to support **model assessment**.

We have developed a framework that is dedicated to both the integration and the assessment of models. To support the former we designed a modular architecture with a generic repository as a central point, with a common meta model and consistency checks. For the latter we defined a mechanism for information retrieval, namely queries of different types. In our entire approach we focus on the static analysis of models.

The languages for expressing constraints and queries over models are an important part of the model assessment process. Depending on their expressiveness it is possible to cover a wider or a narrower range of constraints and to retrieve more or less information from models.

One of the components of our heterogeneous tool environment for model assessment [1] is a generic analysis tool supporting queries over the model repository. Therefore, we started our case study on Object Constraint Language (OCL, [2,3]). In our study we want to examine all types of queries required by our methodology [4]. The OCL 2.0 provides a new definition and querying mechanisms which extend the expressiveness of this language. As described in [5,6], previous versions of OCL (1.x) were not expressive enough to define all of the operations required by relational algebra (RA) and were not adequate query languages (QLs). The main deficit of previous versions was the absence of the tuples concept. In the current version of OCL, tuples are already supported. Thus all primitive operators [7] needed to obtain full expressiveness of a QL, namely *Union, Difference, Product, Select* and *Project*, can be expressed. This fact encouraged us to use OCL within our framework.

The remainder of the paper is structured as follows. In the next section we give a brief introduction to our methodology. Then, we present exemplary models (section 3.1) on which the case study from section 3.2 relies. In section 4 we present a design of our analysis module and finally, in the last section we draw a conclusion.

2 Model Assessment Framework

In this section, the methodology developed within the *MedFlow* project [4,1] is briefly described. A broader description of the methodology developed for systematic model assessment can be found in [4]. The architecture of our tool and the technologies and standards used for its implementation were described in [1]. The design based on the Eclipse Modeling Framework with a generic analysis tool is described in section 4.

As depicted in Fig. 1 at the topmost level of our architecture three components can be distinguished: a modelling environment, a model data repository, and an analysis tool. In this section, only the main ideas related to OCL application within our framework, which are necessary to understand the examples presented in section 3.2, are described.

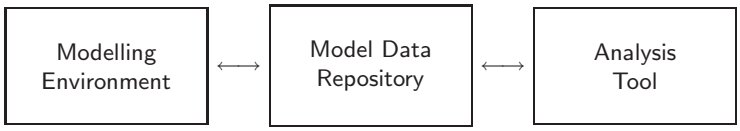


Fig. 1. Base components of our framework

The main assumption in our framework is that all designed models are based on a common **meta model**. Based on the meta model, the constraints for modelling tools are provided and the structure of the common central repository of model elements is generated. **User models** can be imported into the repository from modelling tools via adapters. The usage of a common meta model is crucial for model integration in a heterogeneous modelling environment with diverse notations and modelling tools.

Within our framework we consider two types of OCL expressions: constraints and queries, both defined at the meta model level and evaluated over user models. **Constraints** are related to modelling and extend the specification of models. The aim of using constraints is to support *model consistency* in an early modelling phase. They can be checked automatically each time model elements are saved to the repository or on demand. The expressions used for ensuring syntactical correctness are called *checks* (compare section 3.2). **Queries** are related to the analysis phase and provide aggregated information on sets of model elements. The analysis by means of queries support *semantic quality* of models. As stated in [8], semantic quality belongs to the social layer and needs to be judged by humans. Our framework supports the user in the judgement process by providing mechanisms for information retrieval. Moreover, we can only evaluate the perceived semantic quality comparing user knowledge of the considered domain with his interpretation of models [8] or in our case the results obtained by query evaluations. Both aspects of semantic quality examination — *validity* and *completeness* — can be supported by queries. In the first case we check if all model elements are relevant to the domain. This can be achieved by listing all instances of a given meta model element and human inspection of their relevance. In the second case we look for elements from the domain in the model data repository.

We classify the constraints and queries in four categories (see examples in section 3.2):

Primitive query is the simplest query, which takes as arguments OCL Primitive Types or MOF Classes.

Check is a special kind of primitive query which returns a Boolean value. It is considered as a constraint for a model or, in particular case, as an invariant for a classifier.

Compound query is a query which aggregates results of primitive queries. The arguments of the query are collections. For a given collection the Cartesian Product is built and for each of its element a given primitive query is evaluated. The result is of Set(TupleType) type.

Complementary query is a query evaluated over the result of a given compound query. All other queries are evaluated over a set of model elements. The query can use checks and primitive queries for result calculation.

All types of queries and checks can be evaluated on demand in different scopes selected by a user. We distinguish two types of scopes, namely an evaluation and an initiation scope. The evaluation scope (Fig. 2.a) determines, over what content

the query will be interpreted, and the initiation scope (Fig. 2.b), how the query is called.

Fig. 2. The classification of scopes

Furthermore in both scopes we distinguish modes. In the evaluation scope we distinguish three modes:

Repository mode — the complete set of model elements is considered, i.e., a given expression is evaluated over the content of the repository.

Model mode — only a single user model is considered, e.g. in a running modelling tool on a local machine or model of a predefined type (filtered from the repository). This mode can be used if the queries do not need to be evaluated in the context of the complete set of elements (e.g. checks).

Diagram mode — only one diagram is considered. The usage of this mode is similar to the model mode.

The repository mode is typical for queries in the analysis phase. The advantage of the model and the diagram mode is the possibility of making fast evaluation and ongoing corrections during the modelling phase.

In the initiation scope we consider two different modes, both can be evaluated in any evaluation scope.

Single (element) mode — only queries related to a given element can be activated. This mode enables fine granular analysis of models.

Global mode — all queries can be activated. This mode enables global analysis of models.

An analysis module for our methodology should be generic enough to enable both the definition and interpretation of arbitrary queries. In section 3.2, we examine the expressiveness of OCL and evaluate the possibility of its usage in the analysis module.

3 Case study

In this section the case study from the *MedFlow* project is presented. At first (section 3.1) the excerpt of the domain in question, in form of meta and user

models, is presented. Then examples of queries are presented (section 3.2) and their analysis within our framework is conducted (section 3.3).

3.1 Modelling of clinical processes

In the subsequent sections, parts of a meta model designed within the *MedFlow* project and exemplary user models are presented. The meta model is used as a base for check and query definitions, the user models as a base for check and query evaluations (section 3.2). For our study we used a tool dedicated for OCL compilation, namely the OCL Environment (OCLE, [9]). In this tool, OCL expressions can be compiled and evaluated for single instances or for an entire project. The models and all queries were implemented in the OCLE version 2.0. We stress the fact that the used OCL syntax is the one implemented in the OCLE.

Meta model The aim of the *MedFlow* project was the optimisation of clinical processes. Within the project, we developed a meta model of the clinical processes domain. Fig. 3 shows a fragment of the meta model (the complete meta model can be found in [4]).

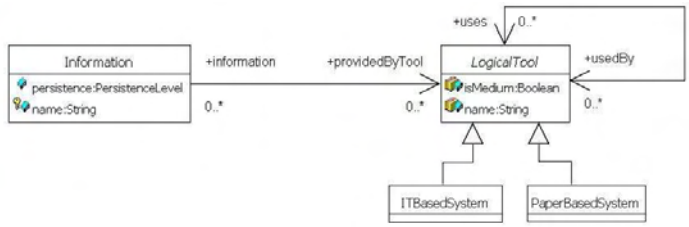


Fig. 3. Part of the *MedFlow*'s meta model

In the meta model excerpt we can distinguish two main classes: Information and LogicalTool. LogicalTool is an abstract class with two subclasses: ITBasedSystem and PaperBasedSystem. Information can be saved in LogicalTool, expressed by an association providedByTool. LogicalTool can use another LogicalTool, what is expressed by the association uses. This simplified meta model is used as a base for the check and query definitions in section 3.2.

User models Based on an meta model (c.f. the previous section) user models are created. In our case study, we used the simplified meta model and two exemplary user models presented in Fig. 4.

In the first user model (Fig. 4.1) four instances of Information and four instances of LogicalTool are defined. The instances of Information have diverse persistence levels (low, medium, high) and instances of LogicalTool are of diverse type (IT–

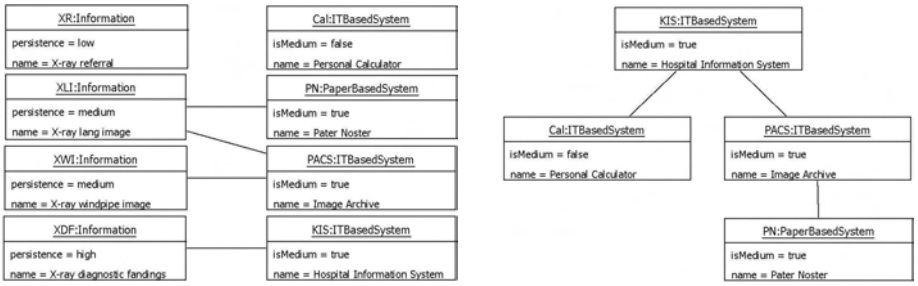


Fig. 4. Exemplary user models: (1) instances of classes Information, LogicalTool and associations between them, (2) hierarchy of instances of LogicalTool

and paper–based). An Information can be saved in a LogicalTool if the LogicalTool is a medium (c.f. Example 2). There are four association links between instances of Information and instances of LogicalTool. In the second user model (Fig. 4.2) the hierarchical dependencies between four instances of LogicalTool are defined. These simplified user models are used as a base for the check and query evaluations in the next section.

3.2 Definition and evaluation of checks and queries

In this section, we present typical checks and queries. All definitions conform to the *MedFlow* meta model (Fig. 3) and their results are evaluated over the exemplary user models (Fig. 4). The examples are based on a representative selection of all types of checks and queries used within our framework for model assessment.

In the examples the checks and queries are defined in natural language and inspected manually. The corresponding listings are expressed in OCL 2.0 and automatically evaluated in OCLE version 2.0.

If not stated divers, definitions (`def`) and invariants (`inv`) are defined and evaluated in the context of Information (`context Information`) and based on the diagram depicted in Fig. 4.1. This context is added for technical reasons to enable easier compilation of OCL expressions. The definitions themselves are not context dependent (no reference of self is used within them).

Primitive query A primitive query can return a value of primitive type (except the Boolean type), class type or collection type. The construction of a primitive query is similar to the below defined examples for checks, thus we do not provide additional examples.

Check The simplest concept for information retrieval is a check. It is a function with a set of objects as a domain.

In Example 1 and Listing 1, the check is defined and evaluated. It checks if there exists an association between a given Information and a given LogicalTool.

Example 1. (theCheck)
Definition: *Is a given information saved in a given logical tool?*
Evaluation for XLI and KIS: no.

Listing 1 (theCheck)
Definition:

```
1  def: let
     theCheck(i:Information, lt: LogicalTool)
3        = i.providedByTool->intersection(Set{lt})->notEmpty()
```

Evaluation:

```
1  def:
     let objInfo         = Information.allInstances
3                          ->select(name="X-ray lung image")
                           ->any(true)
5    let objLTool        = LogicalTool.allInstances
                           ->select(name="Hospital Information System")
7                          ->any(true)
     let theCheckResult = theCheck(objInfo, objLTool)
9  -- Selection: Boolean = false
```

Predefined check Checks can be used to express some well–formedness rules. Such checks should be defined during the meta modelling phase and are called predefined checks.

In Example 2 and Listing 2 a predefined check is defined and evaluated.

Example 2. (thePredefinedCheck)
Definition: *An information can be saved only in logical tools which are mediums.*
Evaluation: is fulfilled for all instances.

Predefined checks can be expressed in the form of invariants and checked for all instances of the context class by calling the function *check UML models for errors* in the OCLE tool.

Listing 2 (thePredefinedCheck)
Definition:

```
1  inv: self.providedByTool->forAll(lt | lt.isMedium=true)
```

Evaluation *check UML models for errors*:

```
Model appears to be correct according to the selected rules.
```

Compound query In order to aggregate information collected with single queries, we can build a compound query. The collections of elements, used as arguments, can be build in different manners, we can use all instances or a subset of them.

Example 3 and Listing 3 depict the results of the compound query with the check defined in Example 1 and Listing 1, applied for all instances of Information and LogicalTool. In Example 3, the result is presented in form of a table while in the Listing 3 it is presented as a set of tuples.

Example 3. (theCompoundQuery)
Definition:
Evaluate `theCheck` *for all instances of* Information *and* LogicalTool *classes.*
Evaluation:

Information \ Logical Tool	KIS	PACS	PN	Cal
XR	no	no	no	no
XLI	no	yes	yes	no
XWI	no	yes	no	no
XDF	yes	no	no	no

Listing 3 (theCompoundQuery)
Definition:

```
1  def: let
     theCompoundQuery(InfC:Set(Information), LToolC:Set(LogicalTool)) :
3      Set(TupleType(
         i : Information,
5        lt: LogicalTool,
         r : Boolean )) =
7          InfC->collect( info | LToolC->collect( ltool |
             Tuple {
9              i : Information = info,
               lt: LogicalTool = ltool,
11             r : Boolean     = theCheck(info, ltool)
           }))->asSet()
```

Evaluation:

```
def:
2  let theCompoundQueryResult =
     theCompoundQuery(Information.allInstances,LogicalTool.allInstances)
4  /*
Selection: Set(Tuple(i:Information,lt:LogicalTool,r:Boolean)) = Set{
6    Tuple{ XDF , PN , false } , Tuple{ XDF , PACS , false } ,
     Tuple{ XDF , Cal , false } , Tuple{ XDF , KIS , true } ,
8    Tuple{ XR , PN , false } , Tuple{ XR , PACS , false } ,
     Tuple{ XR , Cal , false } , Tuple{ XR , KIS , false } ,
10   Tuple{ XWI , PN , false } , Tuple{ XWI , PACS , true } ,
     Tuple{ XWI , Cal , false } , Tuple{ XWI , KIS , false } ,
12   Tuple{ XLI , PN , true } , Tuple{ XLI , PACS , true } ,
     Tuple{ XLI , Cal , false } , Tuple{ XLI , KIS , false }
14 } */
```

Filtering We can additionally apply filters before or after evaluating the result of a given compound query.

The filtered compound query presented in Example 4 and Listing 4 is evaluated only for instances of Information and LogicalTool classes, which fulfil additional constraints.

Example 4. (theFilteredCompoundQuery)
Definition:

Evaluate `theCheck` *for instances of* Information*, which have the* persistence *attribute set to* medium *or* high *and instances of* LogicalTool*, which have the attribute* isMedium *equal to* true.

Evaluation:

Information \ Logical Tool	KIS	PACS	PN
XLI	no	yes	yes
XWI	no	yes	no
XDF	yes	no	no

The definition of `theFilteredCompoundQuery` presented in Listing 4 uses the result `theCompoundQuery` from Listing 3. Like in the previous section, the result (`theFilteredCompoundQueryResult`) is presented as a set of tuples.

Listing 4 (theFilteredCompoundQuery)
Definition:

```
def: let
  theFilteredCompoundQuery() = theCompoundQueryResult->select(t |
          (t.i.persistence=#medium or t.i.persistence=#high)
          and t.lt.isMedium = true )
```

Evaluation:

```
def:
  let theFilteredCompoundQueryResult = theFilteredCompoundQuery()
/*
Selection: Set(Tuple(i:Information,lt:LogicalTool,r:Boolean))= Set{
  Tuple{ XDF , PN , false } , Tuple{ XDF , KIS , true } ,
  Tuple{ XDF , PACS , false } , Tuple{ XWI , PN , false } ,
  Tuple{ XWI , KIS , false } , Tuple{ XWI , PACS , true } ,
  Tuple{ XLI , PN , true } , Tuple{ XLI , KIS , false } ,
  Tuple{ XLI , PACS , true } } */
```

Collecting Elements can be collected according to specific properties (e.g. values of slots, existing links). In the example below we collect elements according to the element hierarchy (c.f. Fig. 4.2). We do not construct a complete definition of a compound query, we only demonstrate how to create a collection using a recursive OCL function.

Example 5. (theCollection)
Definition:
Collect all LogicalTool*s used by a given* LogicalTool*.*
Evaluation for KIS: {PN, Cal, PACS}

Listing 5 (theCollection)
Definition:

```
1  context LogicalTool
     def: let
3      getUsedTools(t: LogicalTool) : Set(LogicalTool)
           = t.uses->collect(x|getUsedTools(x))->asSet()->union(t.uses)
```

Evaluation:

```
def:
2    let objLTool = LogicalTool.allInstances
                      ->select(name="Hospital Information System")
4                     ->any(true)
     let LToolC    = getUsedTools(objLTool)
6  -- Selection: Set(LogicalTool) = Set{ PN , Cal , PACS }
```

Complementary query After the evaluation of a compound query, complementary queries can be evaluated over the obtained result.

In Example 6, a complementary query is defined and evaluated.

Example 6. (theComplementaryQuery)
Definition:
Which instances of LogicalTool *are use to save* Information *objects with persistence level* medium.
Evaluation:
$\{PACS, PN\}$

The OCL expression presented below depicts one of the possible ways to express this complementary query. The condition in line 4 corresponds to the filtering condition and the remaining conditions correspond to the iteration over the result of the compound query.

Listing 6 (theComplementaryQuery)
Definition:

```
def: let
2    theComplementaryQuery: Collection( LogicalTool ) =
       LogicalTool.allInstances()
4        ->select( ltool | theCompoundQueryResult
         ->select( t | (t.i.persistence = #medium) and
6                      (t.lt = ltool and t.r = true))->notEmpty()   )
```

Evaluation:

```
def:
2    let theComplementaryQueryResult = theComplementaryQuery
-- Selection: Collection(LogicalTool)= Set{ PACS , PN }
```

One can notice that the usage of compound queries does not simplify OCL expressions for complementary queries. The complementary query defined in Example 6 can be expressed based on the result of the previously defined compound query (`theCompoundQueryResult`) as in Listing 6 or without any definition as in Listing 7. The results in both listings, 6 and 7, are equal. The

expression in Listing 7 seems to be easier and does not depend on any other definitions.

Listing 7 (theComplementaryQueryBis)
Definition:

```
1 LogicalTool.allInstances
    ->collect(ltool | Information.allInstances
3   ->select(i | i.persistence=#medium).providedByTool)->asSet()
```

At this point, the question why compound queries are useful for complementary queries may arise. Let us explain our motivation for the usage of the first variant. In our prototype for the *MedFlow* project we have a common repository for all models. To evaluate a compound query we have to gather information from the repository, which can be located on a remote server. If we define a complementary query based on the result of the compound query, then the evaluation is faster, otherwise for the evaluation of a complementary query we again need to gather information from the repository. Moreover, we can evaluate more complementary queries over the same compound query without further connection to the repository. The second reason for using the variant with compound queries is the modified presentation of the results of complementary queries. With some additional effort the result can be presented as a set of elements in form of highlighted elements in the result of compound query (c.f. Example 7).

Example 7. (theComplementaryQuery)
Evaluation: $\{PACS, PN\}$

Persistence \ Logical Tool	KIS	**PACS**	**PN**	Cal
low	0	0	0	0
medium	0	**2**	**1**	0
high	1	0	0	0

3.3 Summary

We showed how to construct all types of checks and queries used in our framework. The OCL 2.0 is expressive enough to be applied in our framework for model assessment.

The models created in our framework are MOF compliant and as the OCL supports the object oriented paradigm, it is easy to navigate through the object structures and create checks (compare Example 1) and queries. The invariants can be used as consistency checks before saving models to the repository (Example 2). Tuples provide useful mechanisms for the aggregation of information of different types. Using tuples it is possible to evaluate the Cartesian Product of given sets, what was used within our compound queries concept (Example 3). Using the select operation it is possible to filter collections. The select operation can be applied either to the result of a compound query (Example 4) or to a domain of it (for each argument separately). The first manner enables the expression of more complex conditions (e.g. in the form $(e_1.a_1 = v_1^1 \wedge e_2.a_1 = v_2^1) \vee (e_1.a_1 = v_1^2 \wedge e_2.a_1 = v_2^2)$, where e_i denotes an

element i, a_j an attribute j, and v_i^j some value). OCL does not have a built–in operator for transitive closure, but it allows definitions of recursive functions. In Listing 5 used tools are recursively collected in order to represent the transitive closure of the relation defined by uses. Complementary queries can be expressed in OCL in two different manners. The first is based on a previously defined compound query and the second is a definition from scratch. The first one seems to be easier to automate regarding query definition and results presentation.

4 Technical aspects

In the *SQUAM* project we continue development of the system for quality assessment of models started in the *MedFlow* project [1]. In this section we present redesigned architecture of our system which utilises the newest components developed within the Eclipse Modeling Framework (EMF[1]). The architecture presented below integrates three components of Eclipse Modeling Framework Technology (EMFT[2]), namely Connected Data Objects (CDO[3]), Object Constraint Language (OCL[4]) and Query (QUERY[5]), to create a system with a central model repository and a generic analysis tool. The architecture of the repository and the management of checks and queries are described in subsequent sections.

4.1 Architecture

As mentioned above the design of the model data repository is based on the EMF and some of the EMFT projects. *EMF is a modelling framework and code generation facility for building tools and other applications based on a structured data model* [10]. The model data repository uses EMF as the meta model, it can save model instances of different EMF meta models (c.f. Fig. 5).

The architecture of the model data repository is based on the client–server paradigm. The repository clients can connect to a relational database management system via CDO, which provides multi user support. The connected clients can search, load, save or create new EMF model instances of an arbitrary EMF meta model. Moreover CDO provides a notification mechanism to keep connected clients up to date on model changes.

The repository client integrates the EMFT projects, OCL and QUERY, to specify and execute queries on EMF model elements. OCL component provides an Application Programming Interface (API) for OCL expression syntax which can be used to implement OCL queries and constraints. The QUERY component facilitates the process of search, retrieval and update of model elements; it provides an SQL like syntax.

[1] http://www.eclipse.org/emf/
[2] http://www.eclipse.org/emft/
[3] http://www.eclipse.org/emft/projects/cdo/
[4] http://www.eclipse.org/emft/projects/ocl/
[5] http://www.eclipse.org/emft/projects/query/

The *SQUAM* tool family is based on the above described core functionalities out of the EMF and EMFT projects. The repository client API (CDO, EMF, OCL and QUERY) provides an access mechanism for other tools, mostly modelling tools. The tree–based editors can be generated out of EMF meta model definitions. The native editors are especially useful for the prototyping phase, later on we plan to integrate some graphical editors to create model instances. In the *MedFlow* prototype we integrated the MS Visio[6] and MagicDraw[7] modelling tools. We plan to integrate these two modelling tools as well as editors developed within the Graphical Modeling Framework (GMF[8]) with the *SQUAM* tool family.

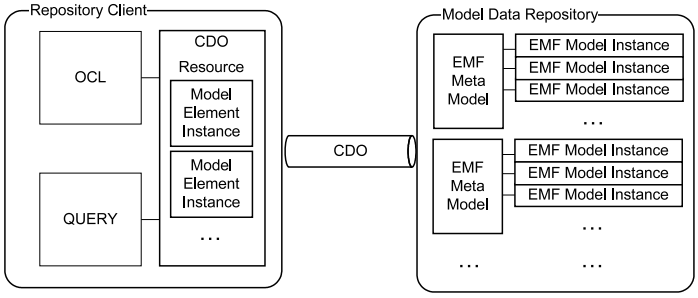


Fig. 5. The model data repository architecture design

For the analysis purposes we use the repository client which uses the OCL component to make queries on the model instances. The management of checks and queries is described in the subsequent section.

4.2 Checks and queries management

The OCL component provides mechanisms for check and query definitions and evaluations. In our framework it should be possible to evaluate checks and queries on demand, thus we need an OCL management system to store OCL expressions. For this purpose we implement a checks and queries catalogue. The catalogue enables users to evaluate OCL expressions in different modes (c.f. Fig. 2 in section 2).

The meta model of the OCL management system is also modelled in EMF, therefore the OCL expressions can also be saved in the model data repository in the same manner as other model instances.

Fig. 6 illustrates the simplified meta model for the OCL management system. The model data repository supports the storage of several meta models. To

[6] `http://office.microsoft.com/visio/`
[7] `http://www.magicdraw.com/`
[8] GMF is a combination of the EMF and GEF (Graphical Editing Framework) projects, `http://www.eclipse.org/gmf/`

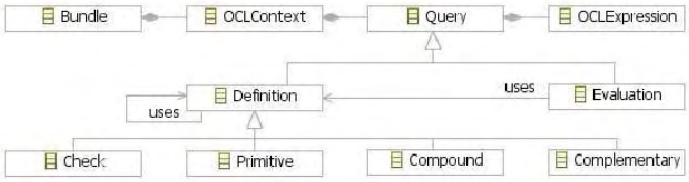


Fig. 6. The meta model of the OCL management system

differentiate between queries specific to a given meta model we assign OCL expressions to a specific Bundle. The Bundle defines the type of the model instances by specifying the meta model they have to conform to.

Further we consider queries, where each Query is placed in a particular OCLContext. The context of the OCL expression enables the usage of the `self` element. The context can also be `NULL`, it is useful for expressions without any particular contexts. In the example listings presented in section 3.2, for all listings except Listing 2 and Listing 5, `NULL` context can be used (these listings do not use the `self` keyword and the definitions are not related to the particular classifier). The Query element contains one OCLExpression.

We distinguish between a definition (Definition) and an evaluation (Evaluation) of queries. Within one Definition the prior definitions can be used, e.g. a compound query can use a primitive query (compare section 3.2). An OCL expression in the Evaluation also uses definitions. The Definition is split into the Check, Primitive, Compound and Complementary expressions. The Definition elements are elements which can be used as subroutines in other expressions and the Evaluation elements are evaluated over an explicit data model, where the OCLContext has to be set to an explicit instance of a model element.

The presented design is a proof of concept for the model data repository. Used technologies and design allow easy extensions with additional features such as dynamic load of new meta models, or an extended editor for the OCL management system with OCL syntax check and compilation at design time.

5 Conclusion

Our examination shows that the OCL is expressive enough to be applied as a query language for model analysis. It is possible to define all types of checks and queries required by our model assessment framework (section 3.2). There are two other reasons for OCL usage within our framework. Firstly, there are more and more tools supporting the OCL notation, also non–commercial tools (e.g. OCL project within EMFT described in section 4 or tools presented in [11]). The second reason ensues from the first: the knowledge of the notation is getting broader among scientists and pragmatic modellers.

We presented a proof of concept for the model data repository created within EMF and EMFT technologies. In the pesented architecture OCL queries for assessment of models can be saved in the repository (section 4.2) and evaluated

on demand. Currently we are developing full support for the OCL management system (section 4.2). We plan to carry out more case studies to determine more requirements for model assessments queries and define patterns for query definitions.

Acknowledgement

We would like thank *Dan Chiorean* for the presentation of the OCLE tool at our University and the later helpful tips for OCL expression implementation in the OCLE. We would like to thank all of the people who reviewed our paper and gave us constructive input, especially *Frank Innerhofer–Oberperfler* and both *reviewers*. And at last but not least *Ruth Breu*, who supported us in our work.

References

1. Chimiak-Opoka, J., Giesinger, G., Innerhofer-Oberperfler, F., Tilg, B.: Tool–supported systematic model assessment. Volume P–82 of Lecture Notes in Informatics (LNI)—Proceedings., Gesellschaft fuer Informatik (2006) 183–192
2. OMG: Object Constraint Language Specification, version 2.0 (2005)
3. Warmer, J., Kleppe, A.G.: The Object Constraint Language—Precise Modeling with UML. first edn. (1999)
4. Breu, R., Chimiak-Opoka, J.: Towards systematic model assessment. In Akoka, J., et al., eds.: Perspectives in Conceptual Modeling: ER 2005 Workshops CAOIS, BP-UML, CoMoGIS, eCOMO, and QoIS, Klagenfurt, Austria, October 24-28. Volume 3770 of Lecture Notes in Computer Science., Springer-Verlag (2005) 398–409
5. Akehurst, D.H., Bordbar, B.: On querying UML data models with OCL. In Gogolla, M., Kobryn, C., eds.: UML. Volume 2185 of Lecture Notes in Computer Science., Springer (2001) 91–103
6. Mandel, L., Cengarle, M.V.: On the expressive power of OCL. In Wing, J.M., Woodcock, J., Davies, J., eds.: World Congress on Formal Methods. Volume 1708 of Lecture Notes in Computer Science., Springer (1999) 854–874
7. Codd, E.F.: Relational completeness of data base sub-languages. Data Base Systems, Rustin(ed), Prentice-Hall publishers (1972)
8. Krogstie, J., Solvberg, A.: Quality of conceptual models. In: Information systems engineering: Conceptual modeling in a quality perspective. Kompendiumforlaget, Trondheim, Norway (2000) 91–120 (available at http://www.idi.ntnu.no/~krogstie/publications/2003/quality-book/b3-quality.pdf, last checked 2006–08–29).
9. LCI team: Object constraint language environment (2005) Computer Science Research Laboratory, "BABES–BOLYAI" University, Romania.
10. Eclipse Foundation Inc.: Eclipse Modeling Framework homepage, `http://www.eclipse.org/emf` (2006)
11. Baar, T., Chiorean, D., Correa, A.L., Gogolla, M., Hußmann, H., Patrascoiu, O., Schmitt, P.H., Warmer, J.: Tool support for OCL and related formalisms - needs and trends. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of Lecture Notes in Computer Science., Springer (2005) 1–9

Rigorous Business Process Modeling with OCL

Tsukasa Takemura[1,2] and Tetsuo Tamai[2]

[1] Software Development Laboratory, IBM Japan, Ltd.,
Yamato-shi, Kanagawa-ken, Japan,
tsukasa@jp.ibm.com
[2] Graduate School of Arts and Sciences, The University of Tokyo,
Tokyo, Japan,
tamai@graco.c.u-tokyo.ac.jp

Abstract. Recently business process modeling is getting a lot of attention as a predominant technology to bridge Business-IT gaps. UML activity diagram was drastically changed in UML 2.0 to support business process modeling. However, this emerging technology is insufficient to bridge the business-IT gaps as we expected because;

- Business process modeling is not so simple as business persons expected.
- Business process models don't provide enough information about activities' behavior.

In this paper, we show that;

- Rigorous description of business helps derivation of business process models.
- Rigorous business process models help to bridge the business-IT gaps.

We propose to use OCL to describe business and to model business process rigorously. We also show necessity of extension of OCL to express constraints of business process models and propose an extension of OCL for business process modeling.

1 Introduction

Enterprises must change their business processes in response to changes in their business enthronement to survive in the intensely competitive society in these days. When enterprises change their business processes, their IT systems also need to be changed according to the changes in their business processes.

Description about the business process provided by business persons tends to be ambigious and does not provide IT persons with enough information to build/change IT systems. This issue is called "business-IT gap", and this gap makes it difficult to change IT systems rapidly according to changes in the business. It is expected that a new technology to bridge business and IT becomes available.

Recently business process modeling is getting a lot of attention as a predominant technology to bridge Business-IT gaps. Business process modeling is

a method to model activities and their sequence, business objects passed among activities, resources, time, and cost required to perform the activities[3].

This emerging technology is not powerful enough to bridge the business-IT gaps as we expected because business process models provided by business persons are tend to be ambiguous. Many business consultants use presentation tools or drawing tools to depict business processes, and the business processes described in this manner have no formal syntax nor semantics. It is one of the reasons to make business process models ambiguous. This ambiguity makes it difficult for IT persons to understand the business process models and realize them as IT systems.

To bridge the business-IT gaps, business process models need to be more rigorous. To make business process models rigorous, we need to define formal syntax and semantics for business process models. One reason of drastic changes in UML activity diagram in UML 2.0[5] was to support business process modeling. Its foundation was changed from state chart diagram to Petri-net. This change allows us to describe data flows and parallel execution of activities in UML 2.0 activity diagrams.

Despite the change, business process models expressed in UML 2.0 activity diagram is still insufficient to bridge the business-IT gap because behavior of each activity or action is not expressed in the activity diagrams. Activities and actions in a business process model need to be implemented as application programs or Web services and so on in order to realize the business process on IT systems. To implement the activities or the actions, their behaviors need to be expressed in the business process model. Without this information, IT persons have no mean to realize the activities nor the actions on IT systems. OCL is thought as the best way to express such information in UML activity diagrams, however, the current OCL is not ready to be used in UML activity diagrams and we propose to be extended to add such information to business process models.

Present methodologies and notations of business modeling assume that it is not complicated work to model AsIs business processes by using control flows and data flows such as UML activity diagram. In some cases, business processes consist of asynchronous activities repeated with different intervals. It is difficult to model such processes only by connecting activities with control flows and object flows. We need to establish a modeling methodology for such complicated asynchronously repeated processes.

2 Case study

2.1 The company

The company in this case study is in manufacturing industry. It obtains materials from suppliers, builds up products from materials, and delivers products to customers.

When the company receives a customer order from a customer, it returns a promise for the customer order to the customer in order to ensure delivery of

products on the expected delivery date specified in the order. A customer order from customers includes product type, quantity, and expected delivery date. A promise also includes includes product type, quantity, and estimated delivery date. For correlation, a promise has the same ID as its associated customer order. In this paper, this business process is called "Order Promising".

The product is made of some materials obtained from suppliers, and the company issues purchase orders to suppliers and receives promises for them from suppliers to obtain the materials. Fig.1 shows Business Use Case diagram for this company's business process, "Order Promising".

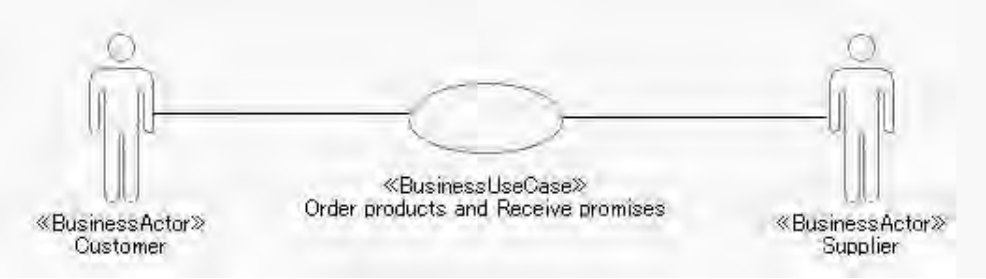


Fig. 1. Business Use Case for Order Promising

This company receives two types of orders; one is "customer order", and another is "forecast". A customer order is a firm order and customers can not cancel it while a forecast shows customer's intention to order products in a specific time frame. The company returns "promise for customer order" when it receives a customer order but does not return promise for "forecast".

The company creates the following three types of plans to prepare materials and satisfy customer orders, and issues purchase orders for materials to satisfy plans. Each plan this company creates includes product type, quantity, and execution complete date.

Resource Plan (RP) Long term plan including productive facilities, allocation of human resources, etc.

Production Plan (PP) Intermediate term plan including provision of long lead time materials

Master Production Schedule (MPS) Short term plan including provision of short lead time materials

The product is made of two types of materials; One is long lead time (LLT) material and another is short lead time (SLT) material. The purchase orders are issued to satisfy the plans.

Fig.2 shows Business Entities in this company and passed between the company and external entities.

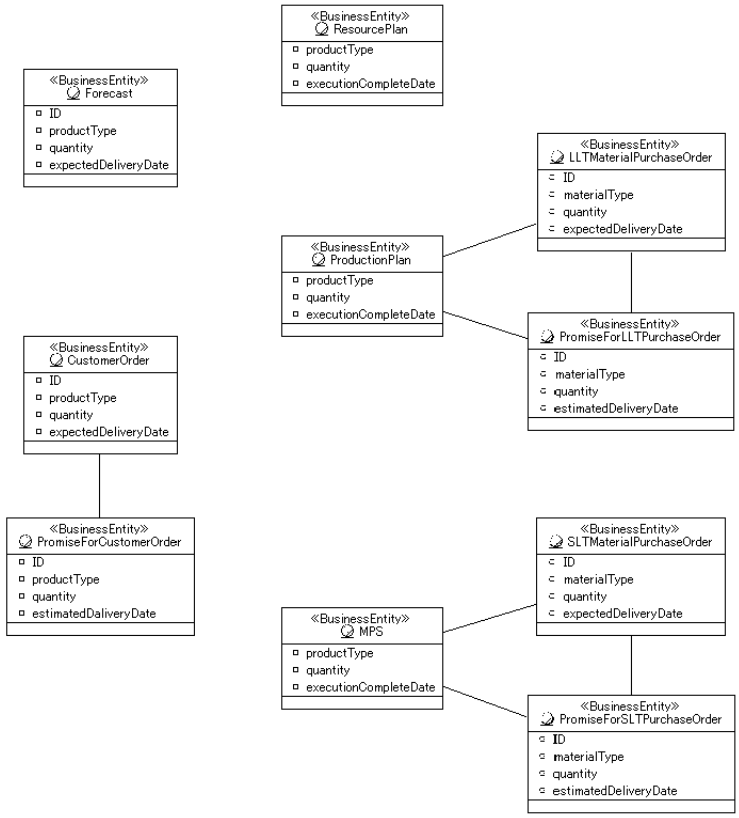


Fig. 2. Business Entities in Order Promising

This business process have predetermined time duration so-called order lead time and planning horizon. Planning horizon is duration from start of execution of a plan to completion of it. Order lead time is duration from issue of a order to expected time of products/materials delivery. Each type of order and plan has a specific lead time or a horizon.

In this case study, we use the following lead times, and planning horizons;

`FORCASTLEADTIME` Forecast lead time, the shortest duration from receiving a forecast to product delivery associated to the forecast.

`CUSTOMERLEADTIME` Customer lead time, the shortest duration from receiving a customer order to product delivery associated to the order.

`LLTMATERIALLEADTIME` Lead time for the long lead time material, the shortest duration from issueing a purchase order for the long lead time material to receiving the material.

`SLTMATERIALLEADTIME` Lead time for the short lead time material, the shortest duration from issueing a purchase order for the short lead time material to receiving the material.

`MPSHORIZON` MPS horizon, the duration from start of execution of a MPS to completion of execution of the MPS.

`PPHORIZON` Production Plan horizon, the duration from start of execution of a production plan to completion of execution of the production plan.

`RPSHORIZON` Resource Plan horizon, the duration from start of execution of a resource plan to completion of execution of the resource plan.

2.2 Order Promising

In this business process, a promise for a customer order is returned in order to ensure delivery of products on the expected delivery date specified in the customer order. To ensure it, a concept, available to promise (ATP) is used in this business domain. ATP is the amount of products available on a specific date including future. It can be calculated by subtracting accumulated amount of promised delivery by the specific date from accumulated amount of products planned to be produced by the specific date(Fig.3). Fig.4 shows ATP conceptually. As is clear from Fig.4, ATP varies on magnitude relation of lead times and planning horizons. From the domain knowledge depict in this figure, it is unveiled that ATP consists of what already provisioned by executed and executing plans and what already committed to promises, and ranges of plans and promises included in ATP are depend on the expected time of delivery.

To ensure delivery of requested products on the expected delivery date, the quantity the company can promise is less than ATP. "Order Promising" is thought as a process to allocate products to a customer order from ATP. In other words, ATP is the least upper bound of a promise for a customer order.

ATP is a key concept of "Order Promising" and it is required for "Order Promising" business process model to express what APT is to transfer the domain knowledge about this process to IT persons.

3 Business modeling

3.1 Business object model

The domain knowledge described rigorously brings out what are inputs to activities and what are outputs from activities. These inputs and outputs can be modeled in a business object model (a kind of class diagram with an UML profile for business process modeling) as shown in Fig.5. In this business object model, classes with ≪BusinessEntity≫ stereotype represent business entities, A class with ≪business process≫ stereotype is not a business entity but an artificial class introduced to specify a context for OCL expressions. By introducing this context in business entity model, the domain knowledge described in the previous section can be expressed in OCL expressions as follows.

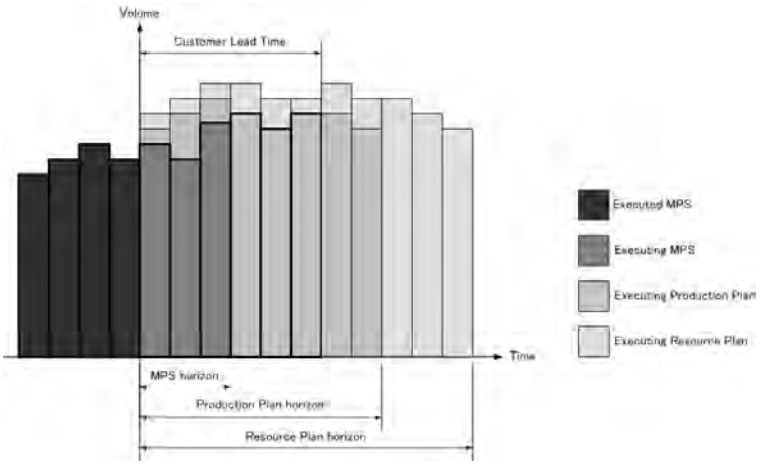


Fig. 3. Rectangles surrounded by bold lines indicate accumulated amount of products planned to be produced within Customer Lead Time

Note that capitalized words in these OCL expressions represent process scoped constant values or operations.

```
context AvailableToPromise::getQuantity( date : Date ) : Integer
body:
existingPlans.getQuantity( date )
- existingPromises.getQuantity( date )

context ExistingPlan::getQuantity( date : Date ) : Integer
body:
mpss->select( executionCompleteDate <= date ).quantity->sum()
+ productionPlans->select( executionCompleteDate <= date and
executionCompleteDate - TODAY() > MPSHORIZON ).quantity->sum()
+ resourcePlans->select( executionCompleteDate <= date and
executionCompleteDate - TODAY() > PPHORIZON ).quantity->sum()

context ExistingPromise::getQuantity( date : Dime ) : Integer
body:
promisesForCustomerOrders->select( estimatedDeliveryDate <= date and
estimatedDeliveryDate - TODAY() > CUSTOMERLEADTIME ).quantity->sum()

context OrderPromising::promiseOrder( )
pre:
newCustomerOrder.expectedDeliveryDate >= TODAY() + CUSTOMERLREADTIME and
// Lead time of the new customer order is longer than CUSTOMERLREADTIME
newPromisesForCustomerOrder->select( ID = newCustomerOrder.ID )->isEmpty()
```

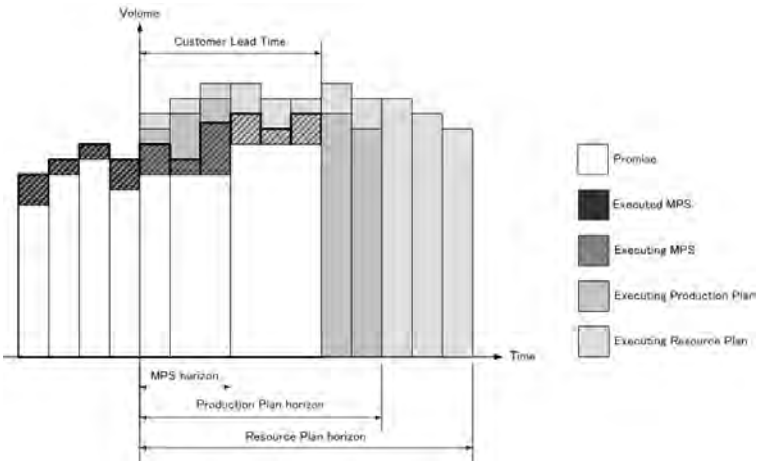


Fig. 4. Rectangles with hatching indicate Available to promise within Customer Lead Time

```
// no promise for the customer order exists
post:
newPromisesForCustomerOrder->select( ID = newCustomerOrder.ID )->notEmpty()
// if there are any promise for the customer order
implies
newPromisesForCustomerOrder->select( ID = newCustomerOrder.ID )->size() = 1 and
// there is only one promise for the customer order and
newPromisesForCustomerOrder->select( ID = newCustomerOrder.ID ).quantity->sum()
= newCustomerOrder.quantity and
// total quantity of promises is equal to requested quantity in the customer order and
newPromisesForCustomerOrder->select( ID = newCustomerOrder.ID )->
forAll( estimatedDeliveryDate = newCustomerOrder.expectedDeliveryDate ) and
// estimated delivery date of the promise is equal to
// expected delivery date of the customer order and
newPromisesForCustomerOrder->select( ID = newCustomerOrder.ID )->
forAll( quantity <= availableToPromise@pre.getQuantity( estimatedDeliveryDate ) )
// total quantity of promises is equal to or less than ATP
```

3.2 Business process model with constraints

When the business process is modeled in UML activity diagram, "promise order" is modeled as an activity or an action. In this paper, we assume that "promise order" is modeled as an action. From the description in the previous section, we modeled "promise order" as an action with five input pins and one output pin as shown in Fig.6.

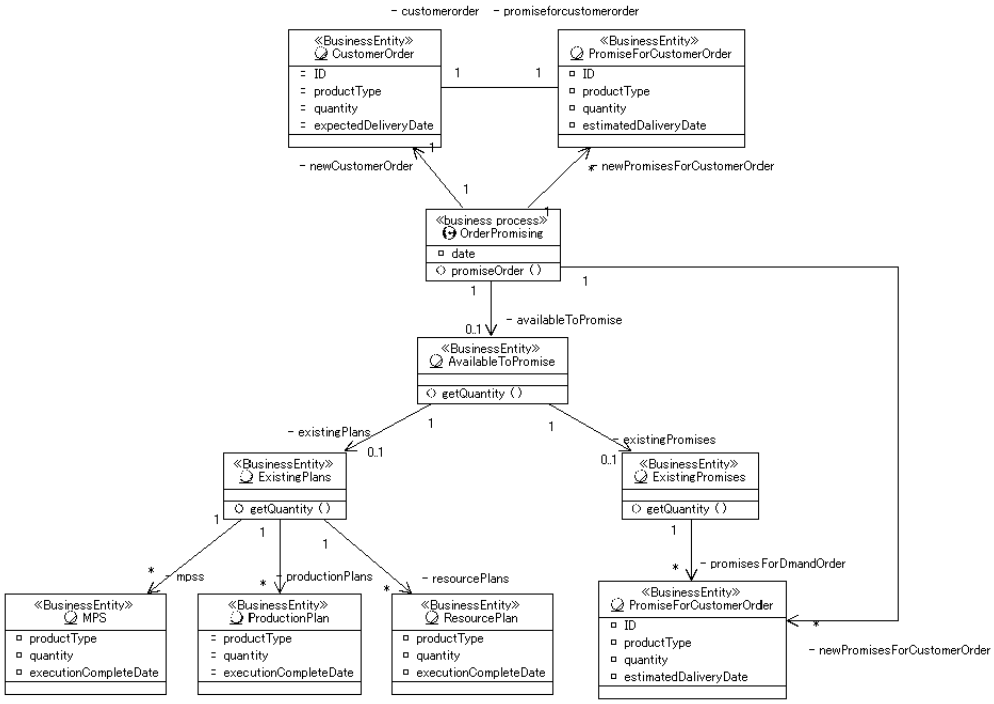


Fig. 5. Business object model for Order Promise

Most commonly, activity diagrams are created prior to a business object model in business process modeling because activity diagrams are more intuitive for business persons like business consultants and their customers.

When an OCL expression is attached to the action, it is required to express the equivalent information as one expressed by OCL attached to the business object model in Fig.5. Especially, we need to describe constraints on output pins by using values of input pins. It means that the OCL expression specifies the action as a context and refers to the pins like as attributes of a class. It requires to extend OCL as we explain in the next section.

3.3 Customization

In this section, we show that the business process can be customized by using constraints defined in the business environment.

Let's say the company in this case study has defined the planning horizons and the lead times as follows;

```
MPSHORIZON < CUSTOMERLAEDTIME < PPHORIZON < FORECASTLEADTIME < RPHORIZON
```

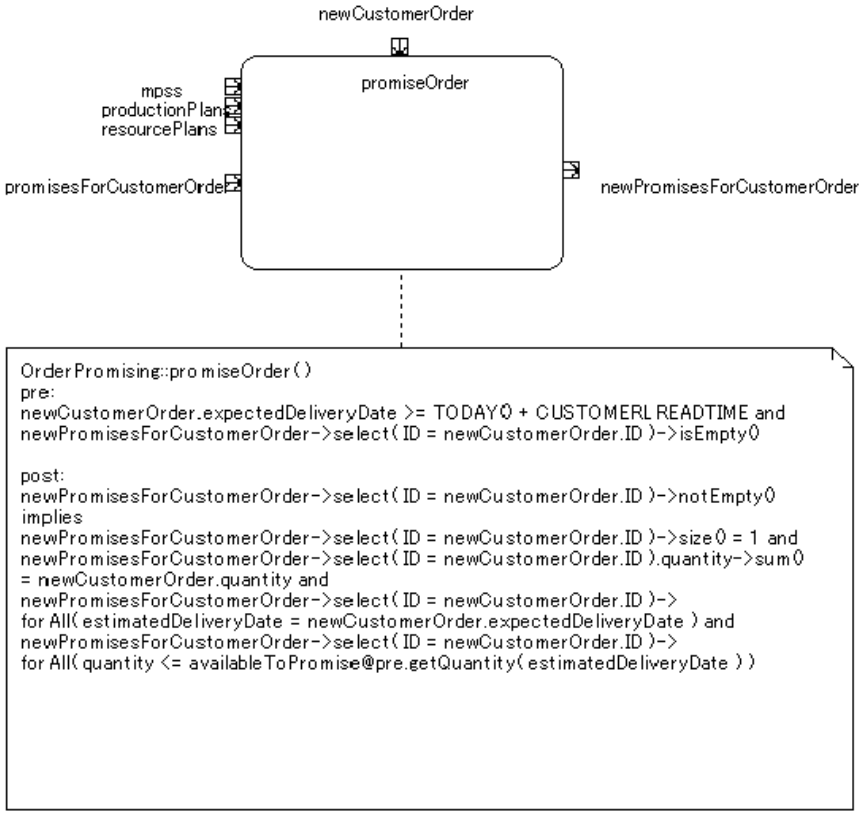


Fig. 6. PromiseOrder action with constraints in activity diagram

By using the magnitude relation `PPHORIZON > CUSTOMERLEADTIME` and an equation `newCustomerOrder.expectedDeliveryDate = TODAY() + CUSTOMERLEADTIME` in the postcondition of promiseOrder, the last term in body of `ExistingPlans.getQuantity` can be reduced as follows;

```
 resourcePlans->select( executionCompleteDate <= date and executionCompleteDate - TODAY()
> PPHORIZON ).quantity->sum()
  ⇓
 resourcePlans->select( executionCompleteDate <= date and executionCompleteDate - TODAY()
> PPHORIZON ).quantity->sum()
  ⇓
 resourcePlans->select( executionCompleteDate <= date and executionCompleteDate > TODAY()
+ PPHORIZON ).quantity->sum()
```

```
  ⇓
 resourcePlans->select( executionCompleteDate <= date and executionCompleteDate > TODAY()
+ CUSTOMERLEADTIME ).quantity->sum()
  ⇓
 resourcePlans->select( executionCompleteDate <= date and executionCompleteDate >
newCustomerOrder.expectedDeliveryDate ).quantity->sum()
  ⇓
 resourcePlans->select( executionCompleteDate <= date and executionCompleteDate > date
).quantity->sum()
  ⇓
 resourcePlans->select( false ).quantity->sum()
  ⇓
 0
```

This reduction indicates that ResourcePlan does not contribute to Available-ToPromise for this action in this business environment, and we can eliminate input pins from promise order action. As in a case of promise order action, some inputs pins and output pins can be eliminated from some other actions. This suggests that constraints for actions help customization of a business process in a specific business environments.

3.4 Composition

Actions modeled in this method have rigorously defined input pins and output pins, these actions can be composed into one business process by connecting pins to common data stores. Fig.7 shows the composed business process model. The gray object flows in this figure denote the object flows connected to the pins eliminated on the customization.

As a result of this composition, a complicated business process where actions are repeated asynchronously can be modeled in an UML activity diagram.

4 OCL extension for business process modeling

As shown in the previous section, OCL is useful for rigorous business process modeling. However, OCL is required to be extended to add the constraints described with OCL to UML activity diagrams. In this section, we explain the required extension for business process modeling using OCL and UML activity diagrams.

4.1 Contexts

To describe behavior of actions, it is required to express preconditions and post-conditions of actions. Consequently, OCL expression needs to have a context from which we can navigate to input pins and output pins of actions in order

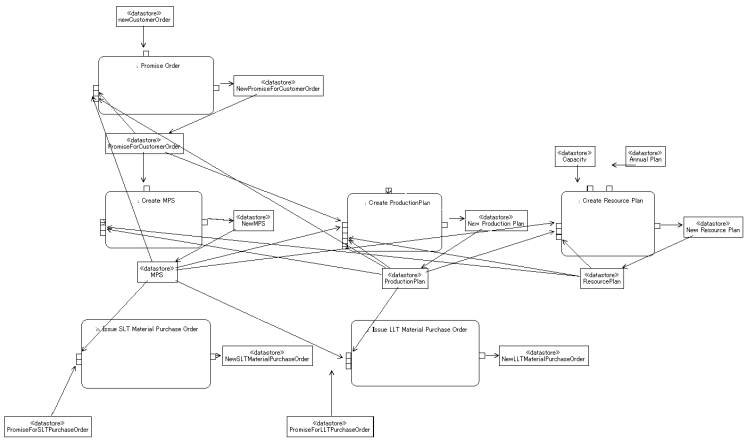


Fig. 7. Composed Business Process Model

to express the constraint of the pins. The action is the best candidate of the context of OCL expression because the action owns the pins.

UML metamodel[5] allows to add Constraint to Actions, however, OCL specification[4] does not expect to specify an action as a context.

The chapter 7.3.4 of OCL specification denotes;

The OCL expression can be part of Precondition or Postcondition, corresponding to ≪precondition≫ and ≪postcondition≫ stereotypes of Constraint associated with an Operation or other behavioral feature. The contextual instance self then is an instance of the type which owns the operation or method as a feature.

First of all, neither Action nor Activity is BehavioralFeature. Even if Constrains are allowed to associate to Actions and Activities, an owner of Actions and Activities is an entity which is executing the business process[8]. The entity executing the business process is a company or an enterprise, and they are not appropriate as a context of OCL expression because we don't model such entities in business process models.

We propose to extend OCL to allow to specify an Action as a context of OCL expression and navigate to its input/output pins.

4.2 @pre

To describe behavior of actions, it is required to express constraints on outputs of actions. Consequently, constraints on objects on output pins need to be expressed by value of objects on input pins. It also required to express the constraints by using the value of inputs on their arrival on the input pins. To refer the value of an input object on arrival on an input pins, it is natural to use "@pre" keyword.

We propose to extend OCL to allow to use the keyword for this purpose.

4.3 Example

By these extensions, we can express constraints on objects on output pins by using value of objects on input pins in the similar way as constraints on operations.

```
context promiseOrder
pre:
newCustomerOrderPin.expectedDeliveryDate >= TODAY() + CUSTOMERLRADTIME and
newPromisesForCustomerOrderPin->select( id = newCustomerOrderPin.ID )->isEmpty()
post:
newPromisesForCustomerOrderPin->select( id = newCustomerOrderPin.ID )->notEmpty()
implies
newPromisesForCustomerOrderPin->select( id = newCustomerOrderPin.ID@pre )->size() =
1 and
newPromisesForCustomerOrderPin->select( id = newCustomerOrderPin.ID@pre ).quantity->sum()
= newCustomerOrderPin.quantity@pre and
newPromisesForCustomerOrderPin->select( id = newCustomerOrderPin.ID@pre )->
forAll( estimatedDeliveryDate = newCustomerOrderPin.expectedDeliveryDate@pre ) and
newPromisesForCustomerOrderPin->select( id = newCustomerOrderPin.ID@pre )->
forAll( quantity <= availableToPromise.quantity@pre( estimatedDeliveryDate ) )
```

5 Related works

As a notation for business process, UML and its extensions are proposed[1][2][7]. Most notations use UML activity diagrams or similar diagram to express activities and their sequence. Modeling methods for these notations assume that persons related to business processes understand the flow of their business process. Our approach does not assume this situation because object flows and control flows of some business process are very complicated and difficult to understand. In our approach, complicated business processes as shown in this paper can be modeled rigorously.

Koubarakis and Plexousakis[6] proposed to model an organization and its business process formally, and showed importance of enterprise model.

6 Conclusion

In this paper, we showed that rigorous description about the business and its related domain knowledge helps creation of business process models. And the created business process models are also rigorously described because preconditions and postconditions of each action are rigorously described by OCL. IT persons can use the preconditions and postconditions to implement the action in the business process on IT systems because the model describes what the action should perform and should not perform formally. If the business process is implemented on Service Oriented Architecture (SOA), the preconditions and postconditions depict requirements for a service.

By describing business environments rigorously, the business process model can be customized systematically. It helps rapid transition of a business process and its IT systems. We can expect the business process models help to bridge Business-IT gaps.

We proposed extension of OCL to support our approach. To describe constraints on actions in UML activity diagram, it is appropriate to specify an action as a context of OCL expression. To support this notation, we proposed extension of OCL.

As shown in this paper, OCL is a powerful tool for business process modeling. But, its syntax and semantics are still difficult to learn for business persons. To resolve this issue,dijkman2002algorithm we are planning to define domain specific languages to simplify OCL for business persons.

References

1. Nuno Castela, Jose M. Tribolet, Alberto Silva, and Arminda Guerra. Business process modeling with UML. In *3rd International Conference on Enterprise Information Systems*, Vol. 2, pp. 679–685, Setubal, Portugal, July 2001.
2. Hans-Erik Eriksson and Magnus Penker. *Business Modeling With UML Business Patterns at Work*. John Wiley & Sons Inc., 2000.
3. Jaap Gordijn, Hans Akkermans, and Hans van Vliet. Business modelling is not process modelling. In *Conceptual modeling for e-business and the web (ECOMO-2000)*, pp. 40–51, Salt Lake City, USA, October 2000. Springer-Verlag.
4. Object Management Group. Ocl 2.0 specification version 2.0. `http://www.omg.org/cgi-bin/doc?ptc/05-06-06`, 2005.
5. Object Management Group. Uml 2.0 superstructure specification. `http://www.omg.org/cgi-bin/doc?formal/05-07-04`, 2005.
6. Manolis Koubarakis and Dimitris Plexousakis. A formal model for business process modeling and design. In *Conference on Advanced Information Systems Engineering*, pp. 142–156, 2000.
7. Chris Marshall. *Enterprise Modeling with UML:Designing Successful Software Through Business Analysis*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.
8. Jos B. Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for Mda*. The Addison-Wesley Object Technology Series. Addison-Wesley, second edition, 2003.

On Interchanging Between OWL/SWRL and UML/OCL

Milan Milanović[1], Dragan Gašević[2], Adrian Giurca[3], Gerd Wagner[3], and Vladan Devedžić[1]

[1] FON-School of Business Administration, University of Belgrade, Serbia
milan@milanovic.org, devedzic@etf.bg.ac.yu
[2] School of Interactive Arts and Technology, Simon Fraser University Surrey, Canada
dgasevic@sfu.ca
[3] Institute of Informatics, Brandenburg Technical University at Cottbus, Germany
Giurca@tu-cottbus.de, G.Wagner@tu-cottbus.de

Abstract. The paper presents a metamodel-driven model transformation approach to interchanging rules between the Semantic Web Rule Language along with the Web Ontology Language (OWL/SWRL) and Object Constraint Language (OCL) along with UML (UML/OCL). The solution is based on the REWERSE Rule Markup Language (R2ML), a MOF-defined general rule language, as a pivotal metamodel and the bi-directional transformations between OWL/SWRL and R2ML and between UML/OCL and R2ML. Besides describing mapping rules between three rule languages, the paper proposes the implementation by using ATLAS Transformation language (ATL) and describes the whole transformation process involving several MOF-based metamodels, XML schemas, and EBNF grammars.

1. Introduction

The benefits of bridging Semantic Web and Model-Driven Architecture (MDA) technologies have been recognized by researchers awhile ago. On one hand, ontologies are a backbone of the Semantic Web defined for sharing knowledge based on explicit definitions of domain conceptualization. The Web Ontology Language (OWL) has been adopted as a de facto language standard for specifying ontologies on the Web. On the other hand, models are the central concepts of Model Driven Architecture (MDA). Having defined a model as a set of statements about the system under study, software developers can create software systems that are verified with respect to their models. Such created software artifacts can easily be reused and retargeted to different platforms (e.g., J2EE or .NET). UML is the most famous modeling language from the pile of MDA standards, which is defined by a metamodel specified by using Meta-Object Facility (MOF), while MOF is a metamodeling language for specifying metamodels, i.e. models of modeling languages. Considering that MDA models and Semantic Web ontologies have different purposes, the researchers identified that they have a lot in common such as similar language constructs (e.g., classes, relations, and properties), very often represent the same/similar domain, and use similar development methodologies [25]. The bottom line is the OMG's Ontology Definition Metamodel (ODM) specification that defines

an OWL-based metamodel (i.e. ODM) by using MOF, an ontology UML profile, and a set of transformations between ODM and languages such as UML, OWL, ER model, topic maps, and common logics [17]. In this way, one can reuse the present UML models when building ontologies.

In this paper, we further extend the research in approaching the Semantic Web and MDA by proposing a solution to interchanging rules between two technologies. More specifically, we address the problem of mapping between the Object Constraint Language (OCL), a language for defining constrains and rules on UML and MOF models and metamodels, and the Semantic Web Rule Language (SWRL), a language complementing the OWL language with features for defining rules. In fact, our proposal covers the mapping between OCL along with UML (i.e., UML/OCL) and SWRL along with OWL (OWL/SWRL). The main idea of the solution is to employ the REWERSE Rule Markup Language (R2ML) [11], [12], [13] (a MOF-defined general rule language capturing integrity, derivation, production, and reaction rules), as a pivotal metamodel for interchanging between OWL/SWRL and UML/OCL. This means that we have to provide a two way mappings for either of two rule languages with R2ML. The main benefit of such an approach is that we can actually map UML/OCL rules into all other rule languages (e.g., Jess, F-Logic, and Prolog) that have mappings defined with R2ML. Since various abstract and concrete syntax are used for representing and sharing all three metamodels (e.g., R2ML XMI, R2ML XML, OWL XML, OCL XMI, UML XMI, OCL text-based syntax), the implementation is done by using Atlas Transformation Language (ATL) [20] and by applying the metamodel-driven model transformation principle [21].

2. Motivation

In this section, we give a simple example of sharing OWL/SWRL and UML/OCL rules, in order to motivate our work. Let us consider an example of a UML model representing relations between members of a family. For a given class *Person*, we can define a UML association with the *Person* class itself modeling that one person is a parent of another one. This is represented by the *hasFather* association end. In the similar way, we can represent relations that one person has a brother by adding another association with the *hasBrother* association end to our model. However, if we one wants to represent that a person has an uncle, i.e. the *hasUncle* association end, this should be derived based on *hasFather* and *hasBrother* association ends, by saying if a person has a father, and the father has a brother, then the father's brother is an uncle of the person. This can be expressed by the UML class diagram and OCL-text based concrete syntax as it is shown in Fig. 1.

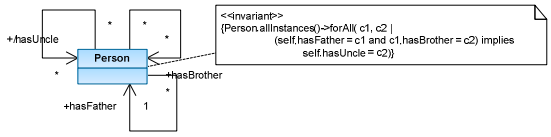


Fig. 1. The family UML model and a OCL invariant on the *Person* class

The same model can be represented as an OWL ontology consisting of the *Person* class and object properties *hasFather*, *hasBrother*, and *hasUncle* (see Fig. 2a). Like in the UML model where the OCL rule has been defined for the *hasUncle* association end, a SWRL rule has to be defined on the OWL ontology for inferring the value for the *hasUncle* object property. This SWRL rule is given in Fig. 2b.

a)

```
<rdf:RDF>
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Person"/>
  <owl:ObjectProperty rdf:ID="hasUncle">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Person"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="hasFather">
    <rdfs:range rdf:resource="#Person"/>
    <rdfs:domain rdf:resource="#Person"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="hasBrother">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Person"/>
  </owl:ObjectProperty>
</rdf:RDF>
```

b)

```
<ruleml:imp>
  <ruleml:_body>
    <swrlx:individualPropertyAtom swrlx:property="hasParent">
      <ruleml:var>x1</ruleml:var>
      <ruleml:var>x2</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom swrlx:property="hasBrother">
      <ruleml:var>x2</ruleml:var>
      <ruleml:var>x3</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_body>
  <ruleml:_head>
    <swrlx:individualPropertyAtom swrlx:property="hasUncle">
      <ruleml:var>x1</ruleml:var>
      <ruleml:var>x3</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_head>
</ruleml:imp>
```

Fig. 2. The family OWL ontology (a) and a SWRL rule in the XML concrete syntax (b)

Even from this rather simple example, one can easily recognize many different languages that are directly involved in the process of interchanging OWL/SWRL and UML/OCL. Given the solution based on the use of the R2ML metamodel as a pivotal metamodel, we can identify the following languages: i) OWL and SWRL abstract syntax, OWL and SWRL XML syntax, and OWL RDF/XML syntax is used for OWL/SWRL; ii) R2ML abstract syntax, R2ML XMI concrete syntax, R2ML XML concrete syntax are used for R2ML; and iii) UML abstract syntax, OCL abstract syntax, UML XMI concrete syntax, OCL XMI concrete syntax, and OCL text-based concrete syntax are used for UML/OCL. Here we advocate a solution that is based on defining mappings between abstract syntax of the three languages where each syntax is represented by MOF, i.e. by a MOF-based metamodel. This means that we can exploit transformation tools (e.g., ATL) for MOF-based model to enable interchange between OWL/SWRL and UML/OCL. In the rest of the paper, we first describe R2ML as the core of our solution, and later we give a full process (conceptual mappings at the level of abstract syntax and implementation details) of transforming between R2ML and OWL/SWRL, and between R2ML and UML/OCL, and thus between OWL/SWRL and UML/OCL.

3. The Interchange Format R2ML

This section is devoted to the description of integrity rules of R2ML [11], [12], [13] developed by the REWERSE WG I1[1] that is used as a basis for interchanging between OWL/SWRL and UML/OCL.

R2ML supports four kinds of rules, namely, integrity rules, derivation rules, production rules, and reaction rules. R2ML covers almost all of the use cases requirements of the W3C RIF WG [27]. Since both SWRL rules and OCL constraints are integrity rules, we just describe R2ML integrity rules here. An *integrity rule*, also known as *(integrity) constraint*, consists of a constraint assertion, which is a sentence

[1] REWERSE Working Group I1–Rule Markup, http://www.rewerse.net/I1

in a logical language such as first-order predicate logic or OCL [6] (see Fig. 3). The R2ML framework supports two kinds of integrity rules: the *alethic* and *deontic* ones. An alethic integrity rule can be expressed by a phrase, such as "*it is necessarily the case that*" and a deontic one can be expressed by phrases, such as "*it is obligatory that*" or "*it should be the case that*."

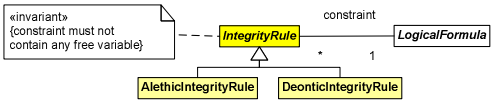


Fig. 3. The R2ML definition of integrity rules

The corresponding *LogicalFormula* must have no free variables, i.e. all the variables from this formula must be quantified. The metamodel of *LogicalFormula* is depicted in Fig. 4. All first order logic constructs for formulas are supported, i.e. conjunctions, disjunctions, and implications.

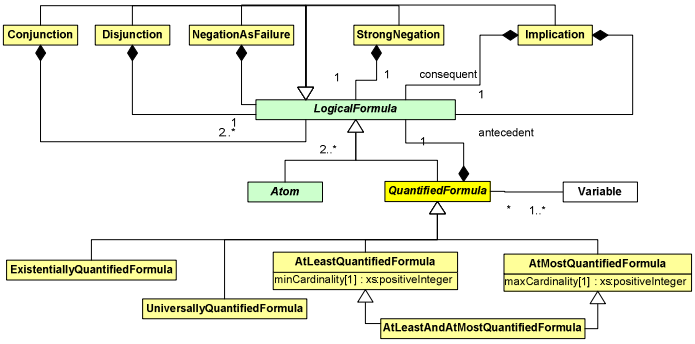


Fig. 4. R2ML logical formula

The distinction between a weak and strong negation is used in several computational languages: it is presented in an explicit form in extended logic programs [4], only implicitly in SQL and OCL, as was shown in [5]. Intuitively, a weak negation captures the absence of positive information, while a strong negation captures the presence of explicit negative information (in terms of Kleene's 3-valued logic). Under the minimal/stable models [3], a weak negation captures the computational concept of negation-as-failure (or closed-world negation) [2].

Quantified formulas, i.e. formulas in which all variables are quantified, represent the core of integrity constraints. Since expressing cardinality restrictions with plain logical formulas leads to cumbersome constructions, R2ML introduces *"at least/most n"* quantified formulas.

Atoms are basic constituents for formulas in R2ML. Atoms are compatible with all important concepts of OWL/SWRL. R2ML distinguishes object atoms (see Fig. 5) and data atoms (see Fig. 6). The design of atoms is tailored to the UML [26] and OCL [6] concepts as well as to OWL [9] and SWRL [7] concepts. Here we present just R2ML atoms necessary for our goal. See [13] for a complete description and use of all supported atoms. An *ObjectClassificationAtom* refers to a class and consists of an object term. Its role is for object classification, i.e. an *ObjectTerm* is an instance of the

referred class. A *ReferencePropertyAtom* associates an object term as *"subject"* with other object term as "*object*." This atom corresponds to the UML concept of object evaluated property, to the concept of an RDF [10] triple with a non-literal object, to an OWL object property, and to the OWL concept of value for an individual-valued property.

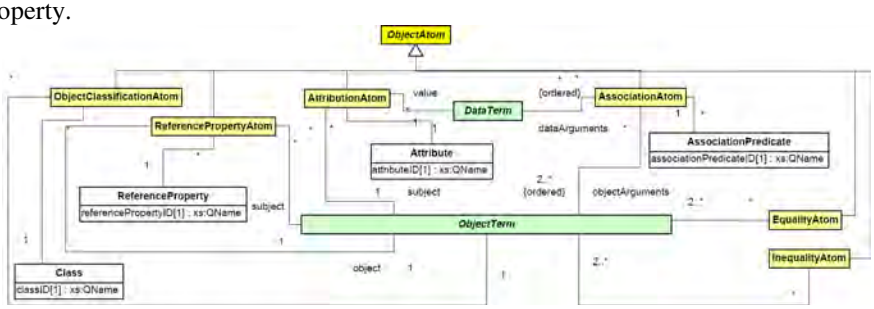


Fig. 5. Object Atoms

An *AttributionAtom* consists of a reference to an attribute, an object term as "*subject*," and a data term as "*value*." It corresponds to the UML concept of attribute and to the OWL concept of value for a data-valued property.

In order to support common fact types of natural language directly, it is important to have *n*-ary predicates (for n > 2). R2ML's *AssociationAtom* is constructed by using an *n*-ary predicate as an association predicate, an ordered collection of data terms as "*dataArguments*," and an ordered collection of object terms as "*objectArguments*." It corresponds to the *n*-ary association concept from UML.

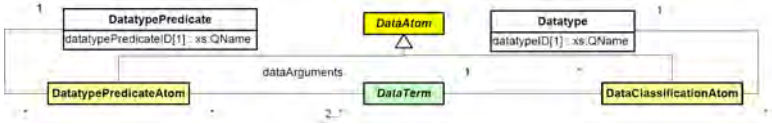


Fig. 6. Data Atoms

R2ML *EqualityAtom* and *InequalityAtom* consist of two or more object terms. They correspond to the *SameIndividual* and *DifferentIndividuals* OWL concepts. An R2ML *DataClassificationAtom* consists of a data term and refers to a datatype. Its role is to classify data terms. An R2ML *DataPredicateAtom* refers to a datatype predicate, and consists of a number of data terms as data arguments. Its role is to provide user-defined built-in atoms. It corresponds to the built-in atom concept of SWRL.

Terms are the basic constituents of atoms. As well as UML, the R2ML language distinguishes between object terms (Fig. 7) and data terms (see Fig. 8). An *ObjectTerm* is an *ObjectVariable*, an *Object*, or an *object function term*, which can be of two different types:

1. An *ObjectOperationTerm* is formed with the help of a *contextArgument*, a user-defined operation, and an ordered collection of arguments. This term can be mapped to an OCL *FeatureCallExp* by calling an object valued operation in the context of a specific object described by the *contextArgument*.

2. The *RoleFunctionTerm* corresponds to a functional association end (of a binary association) in a UML class model.

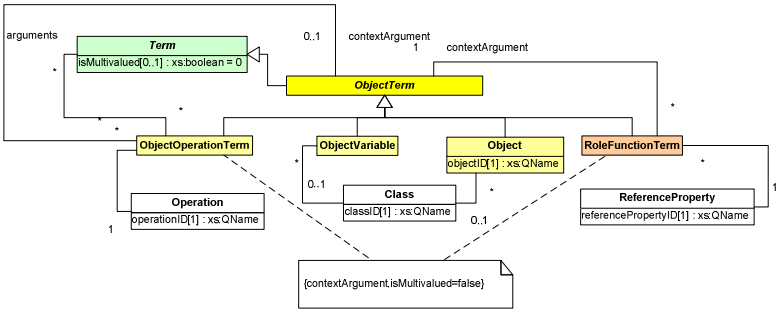


Fig. 7. Object Terms

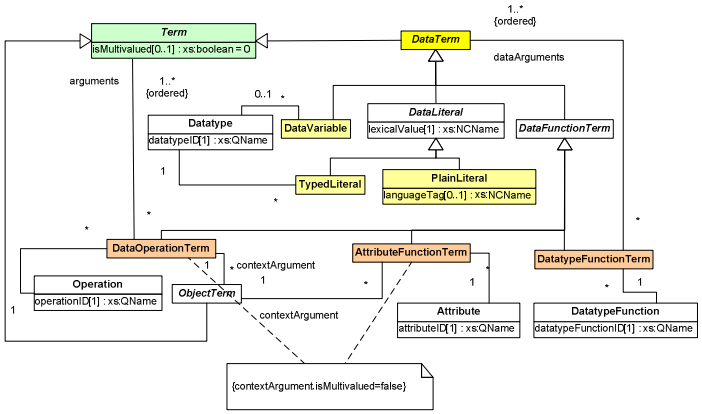


Fig. 8. Data Terms

Objects in R2ML are the same artifacts like in UML. They also correspond to the *Individual* concept of OWL. *Variables* are provided in the form of *ObjectVariable* (i.e. variables that can be only instantiated by objects) and *DataVariable* (i.e. variables that can be only instantiated by data literals).

The concept of data value in R2ML is related to the RDF concept of data literal. As well as RDF, R2ML distinguishes between plain and typed literals (see *DataLiteral* and its subclasses in Fig. 8). They also correspond to the OCL concept of *LiteralExp*.

A *DataTerm* (Fig. 8) is either a data *DataVariable*, a *DataLiteral*, or a *data function term*, which can be of three different types:

1. A *DatatypeFunctionTerm* formed with the help of a user-defined *DatatypeFunction* and a nonempty, ordered collection of *dataArguments*.
2. An *AttributeFunctionTerm* formed with the help of a *contextArgument* and a user-defined *Attribute*.
3. A *DataOperationTerm* formed with the help of a *contextArgument*, a user-defined *operation* that takes as *arguments* an ordered collection of terms.

All of them are useful for the representation of OCL expressions, for example, in *FeatureCallExp* involving data valued operations.

4. Transforming OWL/SWRL to R2ML

In this section, we explain the transformation steps undertaken to transform OWL/SWRL rules into R2ML. In a nutshell, this mapping consists of two transformations. The first one is from OWL/SWRL rules represented in the OWL/SWRL XML format [14] into the models compliant to the RDM (Rule Definition Metamodel) [15]. Second, such RDM-based models are transformed into R2ML models, which are compliant to the R2ML metamodel and this represents the core of the transformation between the OWL/SWRL and R2ML.

The rationale for introducing one more metamodel, i.e. RDM, is that it represents an abstract syntax of the SWRL (with OWL) language in the MOF technical space. As well as SWRL is based on OWL, RDM is also relies on the most recent ODM specification [17]. However, OWL/SWRL is usually represented and used in the XML concrete syntax that is a combination of the OWL XML Presentation Syntax [16] and the SWRL XML concrete syntax [14], i.e. in the XML technical space. However, the RDM metamodel is located in the MOF technical space. To develop transformations between these two rule representations, we should put them into the same technical space. One alternative is to develop transformations in the XML technical space by using XSTL. However, the present practice has demonstrated that the use of XSLT as a solution is hard to maintain [18] [19], since small modifications in the input and output XML formats can completely invalidate the present XSLT transformation. This is especially amplified when transforming highly verbose XML formats such as XMI. On the other hand, we can perform this transformation in the MOF technical space by using model transformation languages such as ATL [20] that are easier to maintain and have better tools for managing MOF-based models. This approach has another important benefit, namely, MOF-based models can automatically be transformed into XMI. We decide to develop the solution in the MOF technical space by using the ATL transformation language. The transformation process consists of three steps as follows. Speaking in terms of ATL, the first step is injection of the SWRL XML files into models conforming to the XML metamodel (Fig. 9). The second step is to create RDM models from XML models, and this process is shown in the right part of Fig. 9 . The third step is transforming such RDM models into R2ML models in the MOF technical space (i.e. the core transformation of the abstract syntax).

Step 1. This step consists of injecting OWL/SWRL rules from the XML technical space into the MOF technical space. Such a process is shown in detail for R2ML XML and the R2ML metamodel in [8]. This step means that we have to represent OWL/SWRL XML documents (Rules.xml from Fig. 9) into the form compliant to MOF. We use the XML injector that transforms R2ML XML documents into the models conforming to the MOF-based XML metamodel that defines XML elements such as XML Node, Element, and Attribute. This XML injector is distributed as a tool along with the ATL engine. The result of this injection is an XML model that can be represented in the XML XMI format, which can be later used as the input for the ATL

transformation. We start our transformation process from the SWRL rule defined in Section 2 and shown in the OWL/SWRL XML concrete syntax (Fig. 2). Fig. 10 shows the XML model which is injected from the SWRL given in Fig. 2.

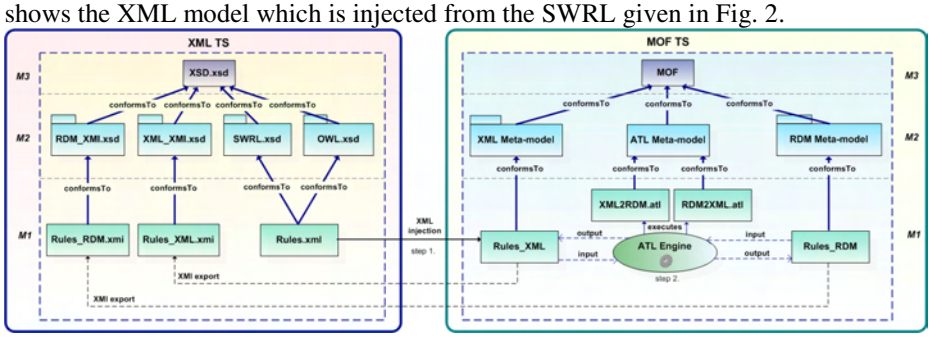


Fig. 9. The first and second steps in the transformation scenario: the OWL/SWRL XML format into the instances of the RDM metamodel

```
<XML.Element xmi.id = 'a8' name = 'swrlx:individualPropertyAtom'
 value = ''>
   <XML.Element.children>
      <XML.Attribute xmi.id = 'a9' name = 'swrlx:property'
       value = 'hasUncle'/>
      <XML.Element xmi.id = 'a10' name = 'ruleml:var' value = ''>
         <XML.Element.children>
            <XML.Text xmi.id = 'a11' name = '#text' value = 'x1'/>
         </XML.Element.children>
      </XML.Element>
      <XML.Element xmi.id = 'a12' name = 'ruleml:var' value = ''>
         <XML.Element.children>
            <XML.Text xmi.id = 'a13' name = '#text' value = 'x3'/>
         </XML.Element.children>
      </XML.Element>
   </XML.Element.children>
</XML.Element>
```

Fig. 10. The IndividualPropertyAtom from Fig. 2 as an instance of the XML metamodel in its XMI format

Step 2. In this step, we transform the XML model (*Rules XML* from Fig. 9) into the RDM-compliant model (*Rules RDM* from Fig. 9). This transformation is done by using the ATL transformation named *XML2RDM.atl*. The output RDM model (*Rules RDM*) conforms to the RDM metamodel. An excerpt of the RDM model for the rule from Fig. 2 is shown in Fig. 11. It is important to say that we can not exploit the standardized QVT transformation between UML and OWL from [17], since our input rules are a combination of SWRL and OWL (i.e., RDM nad ODM).

In the XML2RDM.atl transformation, source elements from the XML metamodel are transformed into target elements of the RDM metamodel. The XML2RDM.atl transformation is done on the M1 level (i.e. the model level). This transformation uses the information about elements from the M2 (metamodel) level, i.e., metamodels defined on the M2 level (i.e., the XML and RDM metamodels) in order to provide transformations of models on the level M1. It is important to point out that M1 models (both source and target ones) must be conformant to their M2 metamodels. This principle is well-know as metamodel-driven model transformations [21].

```
<XMI xmi.version = '1.2' timestamp = 'Wed Jul 19 23:01:46 CEST 2006'>
  <!--...-->
    <RDM.IndividualVariable xmi.id = 'a1' name = 'x2'/>
    <RDM.IndividualVariable xmi.id = 'a2' name = 'x3'/>
    <RDM.IndividualVariable xmi.id = 'a3' name = 'x1'/>
    <RDM.Antecedent xmi.id = 'a4'>
      <RDM.Antecedent.containsAtom>
        <RDM.Atom xmi.idref = 'a5'/>
        <RDM.Atom xmi.idref = 'a6'/>
      </RDM.Antecedent.containsAtom>
    </RDM.Antecedent>
    <RDM.Consequent xmi.id = 'a7'>
      <RDM.Consequent.containsAtom>
        <RDM.Atom xmi.idref = 'a8'/>
      </RDM.Consequent.containsAtom>
    </RDM.Consequent>
    <RDM.Atom xmi.id = 'a5' name = 'IndividualPropertyAtom'>
      <!--...-->
    </RDM.Atom>
   <!--...-->
    <RDM.ODM.ObjectProperty xmi.id = 'a9' name = 'hasBrother' deprecated = 'false'
     functional = 'false' transitive = 'false' symmetric = 'false'
     inverseFunctional = 'false'
      complex = 'false'/>
   <!--...-->
    <RDM.ODM.Rule xmi.id = 'a13'>
       <RDM.ODM.Rule.hasConsequent>
          <RDM.Consequent xmi.idref = 'a7'/>
       </RDM.ODM.Rule.hasConsequent>
       <RDM.ODM.Rule.hasAntecedent>
          <RDM.Antecedent xmi.idref = 'a4'/>
       </RDM.ODM.Rule.hasAntecedent>
     </RDM.ODM.Rule>
  </XMI.content>
</XMI>
```

Fig. 11. The RDM XMI representation of the rule shown in Fig. 2

Step 3. The last step in this transformation process is the most important transformation where we transforming RDM model to R2ML model (Fig. 12). This means that this step represents the transformation of the OWL/SWRL abstract syntax into the R2ML abstract syntax.

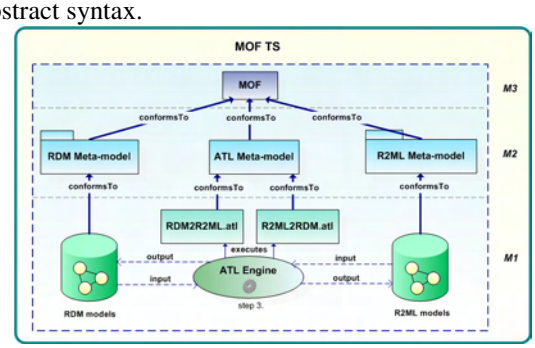


Fig. 12. The transformation of the models compliant to the RDM metamodel into the models compliant to the R2ML metamodel

This transformation step is fully based on the conceptual mappings between the elements of the RDM and R2ML metamodel. In Table 1, we give an excerpt of mappings between the SWRL XML Schema, XML metamodel, RDM metamodel and R2ML metamodel. Due to the size limitation for this paper, we selected a few characteristic examples of mapping rules. The current mapping specification contains 26 rules.

Table 1. An excerpt of mappings between the OWL/SWRL XML schema, XML metamodel, RDM metamodel, and the R2ML metamodel

OWL/SWRL	XML metamodel	RDM metamodel	R2ML metamodel
individualPropertyAtom	Element name = 'swrlx:individualPropertyAtom'	Atom	UniversallyQuantified Formula
OneOf	Element name = 'owlx:OneOf'	EnumeratedClass	Disjunction
var	Element name = 'ruleml:var'	IndividualVariable	ObjectVariable
sameIndividualAtom	Element name = 'swrlx:sameIndividualAtom'	Atom	EqualityAtom
maxcardinality	Element name = 'owlx:maxcardinality'	MaxCardinality Restriction	AtMostQuantifiedFormula

For XML Schema complex types, an instance of the XML metamodel element is created through the XML injection described in Step 1 above. Such an XML element is then transformed into an instance of the RDM metamodel by using ATL, and then to instances of R2ML metamodel.

The actual transformation between the RDM metamodel and elements of the R2ML metamodel are defined as a sequence of rules in the ATL language (*RDM2R2ML.atl* in Fig. 12). These rules use additional helpers in defining mappings. Each rule in the ATL has one input element (i.e., an instance of a metaclass from a MOF based metamodel) and one or more output elements. ATL in fact instantiate the R2ML metamodel (M2 level), i.e. it creates R2ML models. In this ATL transformation, we use so-called ATL matched rules. A matched rule matches a given type of a source model element, and generates one or more kinds of target model elements. Fig. 13 gives an example of a matched rule which is, in fact, an excerpt of the *RDM2R2ML.atl* transformation for the IndividualPropertyAtom class of the RDM metamodel.

```
rule IndividualPropertyAtom2UniversallyQuantifiedFormula {
        from i : RDM!Atom (
                        i.name = 'IndividualPropertyAtom'
                )
        to
                o : R2ML!UniversallyQuantifiedFormula (
                                variables <- i.terms,
                                formula <- refpropat

                        ),
                refpropat : R2ML!ReferencePropertyAtom (
                                referenceProperty <- refprop,
                                subject <- i.terms->first(),
                                object <- i.terms->last()
                        ),
                refprop : R2ML!ReferenceProperty (
                                refPropertyID <- i.hasPredicateSymbol.name
                        )
}
```

Fig. 13. An excerpt of the ATL transformation: A matched rule that transforms an RDM IndividualPropertyAtom to an R2ML UniversallyQuantifiedFormula

For example, the R2ML model shown in Fig. 14 is the output of the RDM to R2ML transformation for the RDM model (IndividualPropertAtom) given in Fig. 11. This is actually the end of the transformation between abstract syntax of OWL/SWRL and R2ML.

An additional step (besides the three ones explained in this section) can be to transform rules from R2ML into the R2ML XML concrete syntax. For example, the

R2ML model shown in Fig. 14 can now be transformed to elements of the XML metamodel (R2ML2XML), and then automatically transformed to the R2ML XML concrete syntax by using the XML extractor that is included the ATL engine. The result of the XML extraction of the R2ML model (from Fig. 14) is shown in Fig. 15.

```
<R2ML>
  <!--...-->
  <R2ML.Formulas.UniversallyQuantifiedFormula xmi.id = 'a12'>
    <R2ML.Formulas.QuantifiedFormula.formula>
      <R2ML.RelAt.ReferencePropertyAtom xmi.id = 'a13'
       isNegated = 'false'>
        <R2ML.RelAt.ReferencePropertyAtom.object>
          <R2ML.BasContVoc.ObjectVariable xmi.idref = 'a11'/>
        </R2ML.RelAt.ReferencePropertyAtom.object>
        <R2ML.RelAt.ReferencePropertyAtom.referenceProperty>
          <R2ML.BasContVoc.ReferenceProperty xmi.idref = 'a14'/>
        </R2ML.RelAt.ReferencePropertyAtom.referenceProperty>
        <R2ML.RelAt.ReferencePropertyAtom.subject>
          <R2ML.BasContVoc.ObjectVariable xmi.idref = 'a6'/>
        </R2ML.RelAt.ReferencePropertyAtom.subject>
      </R2ML.RelAt.ReferencePropertyAtom>
    </R2ML.Formulas.QuantifiedFormula.formula>
    <R2ML.Formulas.QuantifiedFormula.variables>
      <R2ML.BasContVoc.ObjectVariable xmi.idref = 'a6'/>
      <R2ML.BasContVoc.ObjectVariable xmi.idref = 'a11'/>
    </R2ML.Formulas.QuantifiedFormula.variables>
  </R2ML.Formulas.UniversallyQuantifiedFormula>
  <!--...-->
</R2ML>
```

Fig. 14. An excerpt of the R2ML XMI representation of the RDM rule shown in Fig. 11

```
<r2ml:Implication>
  <r2ml:consequent>
    <r2ml:UniversallyQuantifiedFormula>
      <r2ml:ObjectVariable r2ml:name="x1"/>
      <r2ml:ObjectVariable r2ml:name="x3"/>
      <r2ml:ReferencePropertyAtom r2ml:refPropertyID="hasUncle">
        <r2ml:subject>
          <r2ml:ObjectVariable r2ml:name="x1"/>
        </r2ml:subject>
        <r2ml:object>
          <r2ml:ObjectVariable r2ml:name="x3"/>
        </r2ml:object>
      </r2ml:ReferencePropertyAtom>
    </r2ml:UniversallyQuantifiedFormula>
  </r2ml:consequent>
  <!--...-->
</r2ml:Implication>
```

Fig. 15. An excerpt of the R2ML XML representation of the SWRL rule shown in Fig. 2

5. Mapping R2ML Integrity constraints to OCL

In previous section, we have shown how one can get a valid R2ML model from any RDM model. The final objective of this section is to explain the transformation of R2ML models (rules) into OCL models [6]. To do so, we have defined mappings for transforming elements of the OCL metamodel into elements of the R2ML metamodel (see Fig. 16). OCL has its own abstract and concrete syntax, and for transformation process we use its abstract syntax defined in the form of a MOF-based metamodel [6]. Since the R2ML and OCL metamodels are both located in the MOF technical space and there is an metamodel for OCL defined in the OCL specification, the transformation by ATL is straightforward in terms of technological requirements, i.e. we do not have to introduce an additional metamodel like we have done with RDM.

Step 1. We transform an R2ML model (*Rules_R2ML* from Fig. 16) into an OCL model (*Rules_OCL*) by using an ATL transformation named *R2ML2OCL.atl*. The

output OCL model (*Rules_OCL*) conforms to the OCL metamodel. In Table 2, we give an excerpt of mappings between the R2ML metamodel and OCL metamodel on which this ATL transformation is based. The current version of the transformation contains 39 mapping rules.

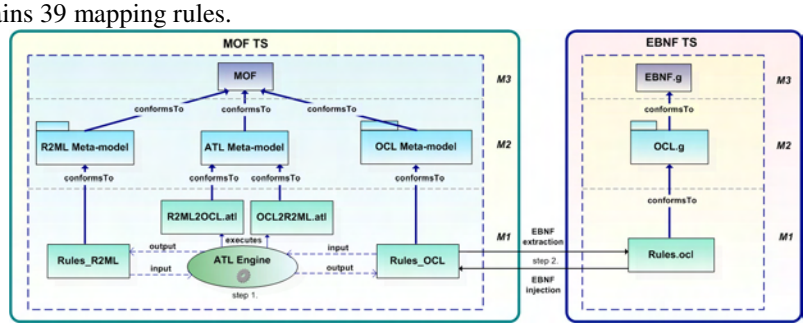


Fig. 16. The transformation scenario: R2ML metamodel to and from OCL metamodel, with EBNF injection/extraction of OCL code

Table 2. An excerpt of mappings between the R2ML metamodel, OCL metamodel, and OCL code

R2ML metamodel	OCL metamodel	OCL code
Conjuction	OperationCallExp referredOperation (name = 'and')	Operand and Operand
Implication	OperationCallExp referredOperation (name = 'implies')	Expression implies Expression
AttributionAtom	OperationCallExp referredOperation (name = '=') PropertyCallExp (subject)	Subject.attribute = value
ObjectVariable	Variable	Variable name
EqualityAtom	OperationCallExp referredOperation (name = '=')	Expression1 = Expression2 and Expression2 = Expression3, ...
RoleFunctionTerm	PropertyCallExp referredProperty (name = 'property') source Variable	Variable.property
AtMostQuantifiedFormula	OperationCallExp referredOperation (name = '<=') argument maxvalue	Expression <= maxvalue

For element of the R2ML metamodel, an instance of the OCL metamodel is created in the model repository. The ATL transformation is done for classes, attributes, and references. For this transformation we have used integrity and derivation rules of the R2ML metamodel in its current version (0.3). For the R2ML model (rule) shown in Fig. 14, we get an OCL model represented in the OCL XMI concrete syntax given in Fig. 17. The figure shows an OCL iterator expression (forAll) that has Boolean as the return type, and an operation call expression as its source. That operation call expression then calls an operation, which has the set type as its return type and a class as its source.

Step 2. Because the OCL concrete syntax is located in the EBNF technical space, we need to get an instance of the OCL metamodel (abstract syntax) into EBNF technical space. There are three possible solutions to this problem. The first one is

creating another transformation from the OCL metamodel to the ATL metamodel (which extends a modified standard OCL metamodel), e.g., to its query expression, and then by using the tool "Extract ATL model to ATL file" included in ATL, we can get the OCL code. However, the disadvantage of this solution is the creation of a new transformation from the OCL metamodel to the ATL metamodel, which is a time consuming and more general task overcoming the scope of our research. The second solution is to create ATL query expression on OCL metamodel elements, which will then generate OCL code in a file. The disadvantage of this solution is that this solution can not be used for all OCL metamodel elements, because it will be complex to define all mappings. The third solution is to use a TCS (Textual Concrete Syntax) interpreter [22] based on the TCS syntax definition of OCL. A TCS represents a domain specific language for the specification of textual concrete syntaxes in MDE, and it is a part of the ATL tool suite. It can be used to parse text-to-model and to serialize model-to-text. The concrete syntax of OCL has been implemented in TCS according to the syntax specified in [6]. Fig. 18 shows the mapping from the OCL metamodel (in the KM3 format [24]) to its corresponding TCS.

```
<OCL.EssentialOCL.IteratorExp xmi.id = 'a11' name = 'forAll'>
   <UML.TypedElement.type>
        <UML.PrimitiveType xmi.id = 'a4' name = 'Boolean'/>
   </UML.TypedElement.type>
   <OCL.EssentialOCL.CallExp.source>
      <OCL.EssentialOCL.OperationCallExp xmi.id = 'a18'>
            <UML.TypedElement.type>
              <OCL.EssentialOCL.SetType xmi.idref = 'a19'/>
            </UML.TypedElement.type>
            <OCL.EssentialOCL.CallExp.source>
              <UML.Class xmi.idref = 'a17'/>
            </OCL.EssentialOCL.CallExp.source>
            <OCL.EssentialOCL.OperationCallExp.referredOperation>
              <UML.Operation xmi.idref = 'a25'/>
                <!--...-->
            <OCL.EssentialOCL.OperationCallExp.referredOperation>
      </OCL.EssentialOCL.OperationCallExp>
   </OCL.EssentialOCL.CallExp.source>
   <!--...-->
</OCL.EssentialOCL.IteratorExp>
```

Fig. 17. An excerpt of the OCL XMI representation of the R2ML model shown in Fig. 14

Using the TCS interpreter and defined mapping rules (as in Fig. 18 for element Class), we have done an EBNF extraction from the OCL model to the OCL code. Our starting example shown in Fig. 1 is actually the OCL code that represents the OCL model from Fig. 17. This OCL code is also the transformed SWRL rule from Fig. 2.

In the opposite direction, from OCL to R2ML, for Step 1 (the ENBF injection), we also have two solutions. The first one is to use the OCL Parser from the Dresden OCL Toolkit [23] for parsing OCL code and creating OCL model from it. This solution needs a predefined UML model (in the UML XMI format) as the input on which OCL code is defined, and this is not what we want, because for the input we want only OCL code without the UML model on which it is defined. The second solution is by using TCS for creating model from code. Since we used this solution for generating code from model and it supports generation of OCL code without UML model, we decided to us it, for this direction. When the OCL model is generated form the OCL code, we use OCL2R2ML.atl transformation for transforming this model into the corresponding R2ML model (as shown in Fig. 16).

```
package OCL {
 //...
   class Class extends Type {
        reference ownedOperation[*] container : Operation;
        reference ownedAttribute[*] container : Property;
        attribute isAbstract : Boolean;
   }
 //...
}
                                a)
syntax OCL {
 //...
   template Class context
        : (isAbstract ? "abstract") "class" name
                "{"
                        ownedOperation ownedAttribute
                "}"
   ;
 //...
}
                                b)
```

Fig. 18. The transformation of elements of the OCL metamodel into its corresponding OCL textual concrete syntax (TCS): a) OCL metamodel; b) OCL TCS

6. Conclusions

The presented approach to interchanging OWL/SWRL and UML/OCL is based on the pivotal (R2ML) metamodel that addresses the complexity of mappings between two languages, which contain many different concepts. In this paper, we have not focused only on mapping rules between OWL/SWRL and R2ML and between UML/OCL and R2ML [13], but we have also described the whole transformation process based on the use of the ATL model transformation language and several other XML schemas and MOF metamodels. Besides bridging OWL/SWRL and UML/OCL, the use of R2ML allows us to reuse (i.e., apply on OWL/SWRL and UML/OCL) the previously implemented transformations between R2ML and R2ML XML concrete syntax, F-Logic, Jess, and RuleML, thus further interchanging OWL/SWRL and UML/OCL.

The presented research is a next step towards the further reconciliation of MDA and Semantic Web languages, and hence continues the work established by the OMG's ODM specification that only addressed mappings between OWL and UML, while we extended it on the accompanying rule languages, i.e., SWRL and OCL. In the future, we finalize the on-going implementation of all transformation proposed in the paper, which will be followed by the detailed report on the experience. We also plan to extend our rule transformation framework in order to support other OMG's specifications covering rules, i.e., the ones for business and production rules.

References

1. Biron, P. V., Malhotra, A. (2004). "XML Schema Part 2: Datatypes Second Edition," W3C Recommendation, http://www.w3.org/TR/xmlschema-2/.
2. Clark, K.L. (1978). "Negation as Failure," *In Gallaire, H., and Minker, J. (eds.), Logicand Data Bases*, Plenum Press, NY, pp.293-322.
3. Gelfond, M., Lifschitz, V. (1988). "The stable model semantics for logic programming," *In Proc. of ICLP-88*, pp. 1070-1080.

4. Gelfond, M., Lifschitz, V. (1991). "Classical Negation in Logic Programs and Disjunctive Databases," *New Generation Computing*, vol. 9, pp. 365-385.
5. Wagner, G. (2003). "Web Rules Need Two Kind of Negations," *In Proc. of the Workshop on Principles and Practice of Semantic Web Reasoning*, pp.33-50.
6. OMG OCL (2006). "Object Constraint Language," *OMG Specification, Version 2.0, formal/06-05-01*, http://www.omg.org/docs/formal/06-05-01.pdf.
7. Horrocks, I., et al. (2004). "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," W3C Member Submission, http://www.w3.org/Submission/SWRL/.
8. Milanović, M., et al. (2006). "Validating Rule Language Metamodels with the Help of Model Transformations," *2nd Int. Conf. of Rules and Rule Markup Languages for the Semantic Web*, Athens, USA (submitted).
9. Patel-Schneider, P. F., Horroks I. (2004). "OWL Web Ontology Language Semantic and Abstract Syntax," http://www.w3.org/2004/OWL.
10. Klyne, G., Carroll J., (eds.) (2004). "Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Rec. 10 February 2004, http://www.w3.org/TR/rdf-concepts/.
11. R2ML (2006). *The REWERSE I1 Rule Language*, http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6.
12. Wagner, G., Giurca, A., Lukichev, S. (2005). "R2ML: A General Approach for Marking-up Rules," *In Proceedings of Dagstuhl Seminar 05371, in F. Bry, F. Fages, M. Marchiori,H. Ohlbach (Eds.) Principles and Practices of Semantic Web Reasoning*, http://drops.dagstuhl.de/opus/volltexte/2006/479/.
13. Wagner, G., Giurca, A., Lukichev, S., Antoniou G., Damasio C. V., Fuchs N. E., "Language Improvements and Extensions," *REWERSE I1-D8 deliverable*, http://rewerse.net/deliverables.html.
14. Horrocks, I., Patel-Schneider, F., P., Boley, H., Tabet, S., Grosof, B., Dean, M. (2004). "SWRL: A Semantic Web Rule Language, Combining OWL and RuleML," *W3C Member Submission*.
15. Brockmans, S., Haase, P. (2006). "A Metamodel and UML Profile for Rule-extended OWL DL Ontologies - A Complete Reference," Universität Karlsruhe (TH) - Technical Report.
16. Hori, M., Euzenat, J., Patel-Schneider, F., P. (2003). "OWL Web Ontology Language XML Presentation Syntax," W3C Note.
17. OMG ODM (2006). "Ontology Definition Metamodel," 6th Revised Submission.
18. Falkovych, K., Sabou, M., and Stuckenschmidt, H. (2003). "UML for the Semantic Web: Transformation-based approaches," *Knowledge Transformation for the Semantic Web, eds., Frontiers in Artificial Intelligence and Applications, vol. 95*, IOS, Amsterdam, pp. 92-106.
19. Jovanović, J., Gašević, D. "XML/XSLT-Based Knowledge Sharing," *Expert Systems with Applications*, Vol. 29, No. 3, 2005, pp. 535-553.
20. ATLAS Transformation Language (ATL), http://www.sciences.univ-nantes.fr/lina/atl.
21. Bézivin, J. (2001). "From Object Composition to Model Transformation with the MDA," *In Proc. of the 39th Int. Conf. and Exh. on Tech. of OO Lang. and Sys.*, pp. 350-355.
22. Jouault, F. (2006). "TCS: Textual Concrete Syntax," *In Proceedings of the 2nd AMMA/ATL Workshop ATLAS group (INRIA & LINA)*, Nantes, France.
23. Dresden OCL Toolkit, Technische Unversität Dresden, Software Engineering Group, http://dresden-ocl.sourceforge.net.
24. Jouault, F., Bézivin, J. (2006). "KM3: a DSL for Metamodel Specification," *In Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, pp. 171-185.
25. Gašević, D., Djurić, D, Devedžić, V., *Model Driven Architecture and Ontology Development*, Springer, Berlin, 2006.
26. OMG, *Unified Modeling Language (UML)* 2.0, Docs. formal/05-07-04 & formal/05-07-05
27. Rule Interchange Format (RIF) use cases and requirements, W3C Working Draft, http://www.w3.org/TR/rif-ucr/

Realizing UML Model Transformations with USE

Fabian Büttner[1] and Hanna Bauerdick[2]

[1] University of Bremen, Computer Science Department, Database Systems Group
[2] University of Bremen, Center of Computing Technologies (TZI), Artificial Intelligence Group

Abstract. The USE (UML-based Specification Environment) tool has been successfully applied for model validation in the past. In our current work, we are enriching the USE specification language with imperative elements. We employ this extension as an assembler to realize UML model (class diagram) transformations with USE in a flexible way: UML transformations are described using a custom abstract language based on object diagram-like patterns. These descriptions are automatically translated into the imperative USE extensions. Our approach aims to provide a flexible instrument to experiment with different transformations and transformation formalisms.

1 Introduction

In the last years, model transformation has become an increasingly important field within software development. At the same time, the notion of a *model* has become more or less a synonym for models in the object-oriented paradigm. Today, the OMG is about to finalize the Query, Views, Transformations (QVT) [OMG06] specification, which aims to provide a set of standardized formalisms to transform one object-oriented model into another one.

We have successfully employed our USE tool [GR02] for validation of static structure models in the past. In our current work, we are employing USE to develop, apply, and validate model transformations. Although QVT is about to be finalized soon, we feel that we still need more evidence on how well it fits for different kinds of transformations. Our approach gives us a flexible instrument to experiment with different transformations and transformation formalisms in a common and accessible environment.

In earlier work, we utilized several ways to describe transformations of structural models [Büt05,BG06]. As a central part, we have been working on a catalog of transformations of OCL annotated UML class diagrams. In our current work, we are now implementing this catalog based on our extension to USE. We started with a very simple transformation language based on object diagram-like patterns and regular expressions. Later, we realized a need for certain extensions to this language to describe our transformation catalog with reasonable effort. Due to the flexibility of our approach, this extensions could be realized quite easily. We motivate the extensions here, too, because we feel that the underlying problems may also occur in other model transformation scenarios.

On the technical level, we did two things: We first added a small imperative OCL-based language to USE which can be used to define operations. We then created a simple transformation language, based on UML diagrams and a few elements of graph transformation [Roz97]. We then implemented a UNIX filter like program which translates our transformations into the simple imperative language. This way, we can employ USE as a "virtual machine" to execute and validate transformations, under (potentially) various transformation formalisms.

Several other approaches to object-oriented model transformation exists. [CH03] provides a general classification. Existing model transformation frameworks include ATL [JK05], the TopMODL initiative [MDFH04], Modelware [Mod], and the graph transformation-based Fujaba [FUJ]. Our work also resembles [MFJ05] and Kermeta [FDVF06] as it adds executability to meta-models.

This paper is structured as follows: In Sect. 2, we introduce our extension to USE and our transformation language. We then take an excerpt (a "Many2One" transformation) of our transformation catalog to illustrate our approach in Sect. 3. After describing Many2One in general, we show how we implemented the transformation in USE. Section 4 concludes this paper.

2 Realizing UML Model Transformations with USE

In a nutshell, we are employing USE to apply transformations to UML models – or more specifically, to UML class diagrams. Although USE directly supports class diagrams (USE specifications are mainly class diagrams with constraints), we represent them as object diagrams of the UML meta-model here. The USE specification is provided by the UML 2.0 meta-model in our current work.

The two basic ideas of our approach are as follows: *1) The USE specification language is extended to support imperative descriptions of operations.* We achieve this by enriching USE with a minimal object-oriented programming language that can be used to modify a system state (i.e., an object diagram). This language resembles the "ImperativeOCL" language of the upcoming QVT specification. We employ this new feature of USE to formulate additional operations with side-effects for the UML meta-model.

2) Transformations are applied to UML models by adding additional transformations objects to their meta-level representation. These transformation objects provide operations that modify the model in the intended way. The operational transformation semantics is located in explicit transformation meta-classes.

The remaining section explains this sketch in more detail.

2.1 Models Everywhere

Figure 1 shows an overall structural picture of our approach. It shows the different involved modeling artifacts, which creates them, and how they are realized using USE.

Starting in the lower left-hand corner, we locate the role of the *modeler*. That is the origin of our initial model, an exemplary PersonCompany class diagram (which we will modify by means of model transformation in Sect. 3).

This class diagram is created as an instance of the UML meta-model (UMLOCL2 in Fig. 1) in USE. As the name suggests, UMLOCL2 actually combines the UML2 and OCL2 meta-models into one meta-model. It is defined as a USE specification (UMLOCL2.use). So far, this specification does not require any of the extensions mentioned above. In addition, an augmented version UMLOCL2withTransformations.use exists which contains further meta-classes which we describe below.

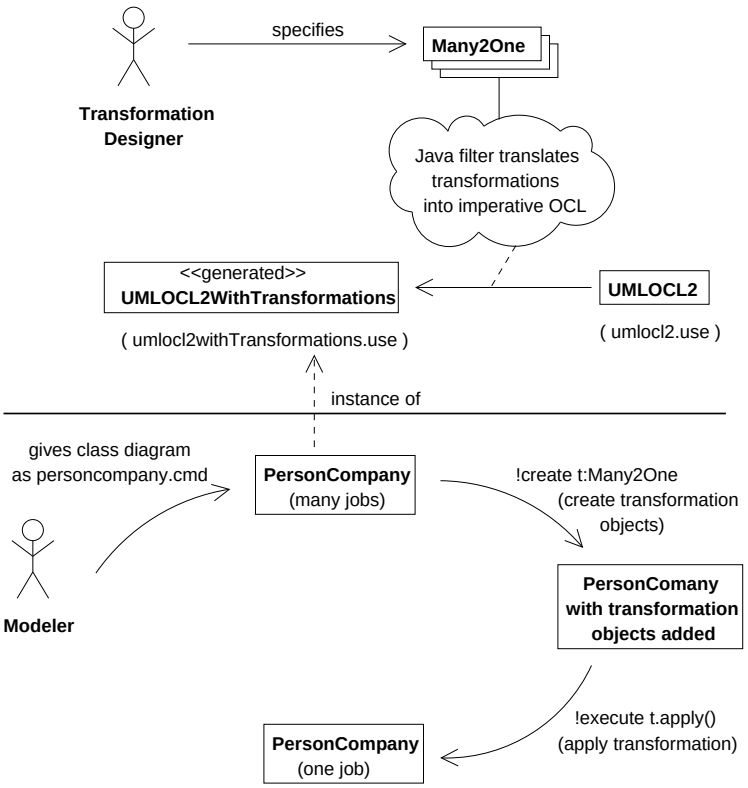


Fig. 1. The overall picture

The modeler creates his or her model in USE as a sequence of state manipulation command (a .cmd-file). After reading and executing this sequence, USE holds the PersonCompany class diagram as an instance of UMLOCL2withTransformations. Of course, manually specifying a UML class diagram as a meta-model instance is not a pleasing task. Therefore we created an import filter, that converts USE specifications (.use-files) into instances of the UML meta-model (.cmd-files that are executable w.r.t. UMLOCL2.use). But this is only for convenience and is not required for our approach. Notice that PersonCompany could also be instantiated as an instance of UMLOCL2, as only

meta-classes from this meta-model are used up to this point. But since we want to enable transformations on PersonCompany, we are using the latter one.

The modeler can now choose one of the transformations that have been previously defined by the *transformation designer* (explained soon). From the modelers point of view this means to add an instance of a corresponding transformation meta-class (Many2One in Fig. 1) to the meta-level representation of his or her class diagram. This transformation meta-classes contain operations with side-effects that realize the intended effect. By convention, each transformation meta-class defines an entry point operation *apply()*. By invoking *apply()* the transformation object changes its surrounding instances of the UML meta-model. This may require several internal steps. Finally, the transformation object is removed again, leaving a meta-level representation of the modified PersonCompany class diagram (for example, with only one job per person now). Alternatively, transformation objects can also be kept for tracing reasons.

How do the transformation meta-classes come into play? First, the transformations are developed by the aforementioned *transformation designer*. In our case study we employ a formalism which consists of simple transformation steps (described as object diagrams with some minor extensions) and regular expressions controlling the correct execution sequence of the individual steps. Each transformation consists of one control expression and one or many transformation steps. Transformations described in this formalism can be visualized in a UML-like representation (extended object diagrams). For our purpose, we have developed a textual syntax, too.

These transformation descriptions, having a high level of abstraction, are then translated into transformation meta-classes using the minimal imperative language we added to USE. As a result we achieve an enriched version of the original UML meta-model. For each transformation class, several internal operations are created to implement the pattern matching and the application of the transformation steps, and the overall control expression. Furthermore, new associations can be added to the meta-model to allow to specify the context for the transformation (e.g., to specify the target association end in the example in Sect. 3). Finally, the existing meta-classes can be enriched by further operations to implement cross-cutting functionality such as cloning or component exchange (described later on).

Currently, this translation (or compilation) from our high level transformation formalism into the imperative language is realized as a Unix filter like Java program. This program takes the original UML meta-model (UMLOCL2.use) and the textual transformation description (ManyToOneTrans.txt) as input files and yields a modified version of the original .use-file. More than one transformation can be added by applying this step repeatedly.

2.2 The imperative extensions to USE specifications

We shortly introduce the new USE specification language concepts. In previous versions of USE, operations could be specified in two ways: a) as OCL queries or b) by providing pre- and postconditions that characterize operation properties.

The first variant is side-effect free. Operations specified by pre- and postconditions are typically not side-effect free, but not automatically executable from USE's point of view neither. They can validated given a manually provided sequence of state manipulation command.

Now we have added a third mechanism to specify operations (with side-effects) in USE. It is a small imperative language that allows us to provide operational specification. It is build around OCL as an expression language. It further adds the following imperative elements:

- Basic state manipulation statements: create and destroy objects, insert and remove links between objects, set attribute values. (These operations were available in USE before in the (still existing) command files.)
- Flow control statements: execute conditionally (if-then-else), execute repeatedly (while), and iterate over collections.
- Invocation of other operations. Other operations can be invoked if they are also defined using this imperative language. Recursive invocation is supported. (Of course, OCL query operations can be used as well, but only in expressions.)

This language resembles the operational mappings language provided by OMG QVT.

2.3 The transformation language used for the catalog

Due to the expressive power of the added imperative language, we can now realize arbitrary transformations on instances. This can be done, as illustrated in the beginning, by putting one or many transformation operations into an explicit transformation class. But writing transformations on this level is still a tedious task for complex transformations. Several higher level alternatives exist that are more appropriate to this task, such as graph transformation or relation-based approaches like in QVT.

Therefore, we created a small, more abstract transformation language to formulate our UML class diagram transformation catalog. Instead of implementing this transformation language in USE, too, we defined a mapping onto the small operational language introduced above. This mapping is currently realized by a small, external piece of software which acts like a Unix filter. It reads an existing source specification and one or many transformation descriptions and creates a specification which is enriched by new *transformation classes* that implement the transformations.

In our transformation language, a transformation consists of two parts: a set of transformation steps and a control expression. The transformation steps build the atoms of a transformation. They are specified as extended UML object diagrams. The control expression then specifies when and in which order the individual steps have to be applied. The control expression language is a regular language whose terminal symbols are the transformation steps.

(Transformation steps) Figure 2 shows a simple transformation step, an extended object diagram. A UML diagram diagram can be regarded as a restricted

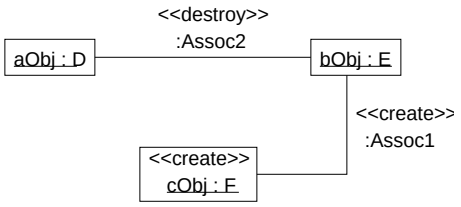


Fig. 2. A simple transformation step

form of a graph transformation rule: Unless marked otherwise, the objects and links in the diagram determine the context of a gratra rule. Elements marked with the stereotype «create» are created by the rule. Elements marked with «destroy» are destroyed by the rule. The destroyed and context elements determine the redex of the rule, i.e. the pattern which must be found in the system state in order to apply the rule. In Fig 2, the redex of the transformation step consists of the objects aObj, bObj, and the Assoc2-link that connects them. In order to apply the step, actual assignments for aObj, bObj and the Assoc2-link have to be found in the system state. Given a certain valid redex, applying the rule will create a new object of class F, connected to bObj, and destroy the Assoc2-link. We further allow OCL preconditions to be part of the extended object diagrams. This way one can further restrict what is a valid redex (not shown in Fig 2).

(Translation of transformation steps) For each step two operations are created in the corresponding transformation class (Many2One in the example which is described in detail in Sect. 3). First a *stepname_redex()* operation is defined which returns a redex for the step. The result type is a tuple consisting of all objects that determine the context of the step. In the above example, the signature is *exampleStep_redex() : Tuple(aObj:D, bObj:E).*[3] Second, a *stepname_apply()* operation is applied which takes a redex tuple and realizes the actual effect of this step. For the example step, this is: *exampleStep_apply(redex:Tuple(aObj:D, bObj:E)).*

(Control expression) A transformation can consists of several steps. The transformation control expression is a regular expression that controls how the steps are combined. An example may look like this:

MyTransform := Init StepA* (StepB | StepC)* CleanUp

The syntactical elements are as usual. Star means as long as possible (include zero times), parenthesis group steps, the bar specifies alternatives. Note that we do not allow a recursive definition (i.e., the derivation tree has to be acyclic). However, it is still possible to create infinite loops, due to '*' expressions. It is the transformation developer's responsibility to ensure that the transformation process terminates.

[3] The links are not a part of the redex tuple, because we assume relation semantics for associations. Multiple links of the same association are not allowed between a pair of objects.

To implement our transformation class for a transformation, we create three elements in the class:

1. A *step()* operation that applies one step (if possible). This operation uses the various ..._redex() operations to determine the next applicable step().
2. A *state* attribute which keeps track of the current transformation state (a state of the finite automata created from the regular control expression).
3. An *apply()* operation which simply applies step() as long as possible. This is the entry point to apply the transformation.

3 Case Study

In this section, we show how we implemented the aforementioned transformation catalog with the extended version of USE. Our transformation catalog resembles the refactorings catalog of Martin Fowler [Fow99]. It contains class diagram transformations such as moving methods from one class to another, changing generalizations into compositions, modifying associations, inlining, and extracting classes.

The major feature of our catalog is that it considers OCL annotations on the UML class diagrams. Because OCL expressions depend on the underlying class diagram, they have to be incorporated when changing its structure. Of course on this account, the transformations become more complex.

Due to this complexity our catalog provides a good example to validate an approach to model transformation (we think). This section provides an exemplary insight into the catalog and its realization in USE. We pick one transformation (Many2One) which changes an association multiplicity from many to one. In the following, we first explain Many2One in more detail, independent of its realization in USE. Then we describe Many2One by means of the previously explained meta-model transformation steps and its control condition. Finally, we illustrate how one of these steps is translated into a USE specification, i.e., into imperative operations.

3.1 Example Transformation: Many2One

Transforming an association multiplicity from *many* to *one* is conceptionally simple but still interesting: a simple modification to the class diagram part has some more demanding consequences on those existing OCL formulas that depend on the modified association.

In OCL, every expression is typed. The type determines which operations can be applied to its values. When navigating through an association end in an OCL expression, the type of this expression is determined by the end's multiplicity. If the multiplicity is `1` or `0..1`, the navigation expression has an object type. Otherwise, the expression results in a collection.

Consequently, when changing the multiplicity of an association end from *many* to *one*, the navigation expression along this end changes from collection-valued to object-valued. Every OCL expression containing this navigation as

a sub-expression is affected. Accordingly, the Many2One transformation also consider the OCL expression parts when changing a class diagram. The Person-Company class diagram in Fig. 3 illustrates the necessary modifications.

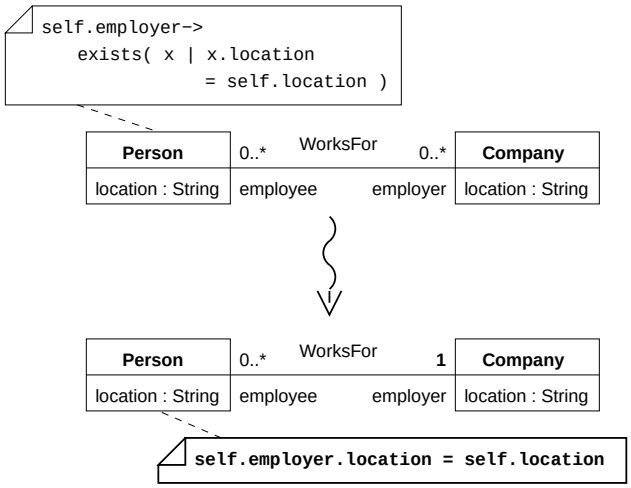


Fig. 3. Transforming an association multiplicity from many to one

Both UML class diagram fractions describe an employer-employee relationship where a person resides in a location and works for a company. The upper invariant states that a person has to work for at least one company which is located in the same place. The depicted model transformation changes the multiplicity at the employer end from arbitrary to one, i.e. that after the transformation every person work in exactly one company. Accordingly, the type of the navigation expression `self.employer` changes from collection-valued to object-valued. Therefore, the invariant should be simplified to the expression stated in the lower part of Fig. 3.

Most implied OCL model transformations are dependent on the performed class diagram transformation. If one for example carries out a class diagram transformation which changes an attribute name, all OCL expressions which use this attribute name have to be adapted. However, if an association multiplicity is changed, more complex OCL expression transformations have to be performed, as explained above.

There is a great benefit if one keeps the number of dependent OCL model transformations to a minimum, because these transformations cannot be reused in other contexts. In case of Many2One, a separation between the dependent and the independent transformations can be realized by following the steps below:

1. Transform all OCL collection operations to `iterate`, if possible.
2. Change the association multiplicity (UML model transformation).
3. Adjust all effected OCL expressions.
4. Simplify the OCL expressions.

Only the second step realizes a transformation of the UML class diagram. The other three transformations refer to OCL expression transformations.

The first step maps all OCL collection operations to `iterate`, which is the most powerful collection operation. For almost every collection operation such a mapping can be defined (one exception e.g. is the `including` operation). The advantage of this mapping is that after the transformation nearly all collection operation expressions are iterate expressions. The number of used collection operations is definitely reduced to a minimum and only for this reduced operation set a corresponding transformation to the UML model transformation has to be found. The most important issue about the first step is that it describes equivalence transformations, i.e. these transformations can be applied in any case and does not change the semantic of the expressions. The same holds for the last step which simplifies the OCL expressions either by performing an inverse transformation to the first one if possible or by applying some basic simplification rules.

Only the third step depends on the UML model transformation and can only be applied if this concrete UML model transformation is performed. The UML class diagram transformation causes that the affected OCL expressions which normally result in a collection become object-valued (they result in exactly one element). Thus within this step, the collection operations have to be transformed to equivalent expressions which only uses object operations.

3.2 Realization of Many2One with USE

All transformations of the case study consist of control expressions and transformation steps. In this section the control expression and the steps of the Many2One transformation are explained.

As mentioned above, the control expression serves as the control structure which coordinates the execution order of the transformation steps. These control expressions are described using regular expressions. The following one describes the process of the Many2One transformation.

```
Many2One = Iteratorize* changeMultiplicity AdjustCollectionOps*
             Simplify*

Iteratorize = existsToIterate | forAllToIterate | ...

AdjustCollectionOps = AdjustIterate | AdjustIncluding | ...

AdjustIterate = inlineRangeForRangeVar inlineAccuInitForAccuVar
                  inlineIterateBodyForIterate

Simplify = iterateToExists | iterateToForAll | ...
```

These control expressions of Many2One are related to the above mentioned transformation steps. `Iteratorize` refers to the transformation of the collection operations to `iterate`. The association multiplicity of the UML class diagram

is changed by the step `changeMultiplicity`. `AdjustCollectionOps` corresponds to the adaptation of the OCL expressions which were affected by the class diagram change. `AdjustIterate` for example transforms an `iterate` expression to an equivalent object-valued expression, if the source expression of iterate refers to exactly one element. This means that during the evaluation of the `iterate` expression always only one iteration is performed. Consequently, the control variables and the result variable can be replaced by their initialization values. These substitution steps are realized by all three `inline` transformation steps. The last step (`Simplify`) is related to the simplification of the OCL expressions. This transformation step realizes the inverse effect of `Iteratorize` and also applies some generic simplification rules.

In the following, some transformation steps of Many2One are exemplarily described to show the functioning of the transformation steps. Figure 4 shows an excerpt of the UML and OCL meta-model which is relevant for the transformation steps of Many2One and consequently fundamental for the understanding of those steps.

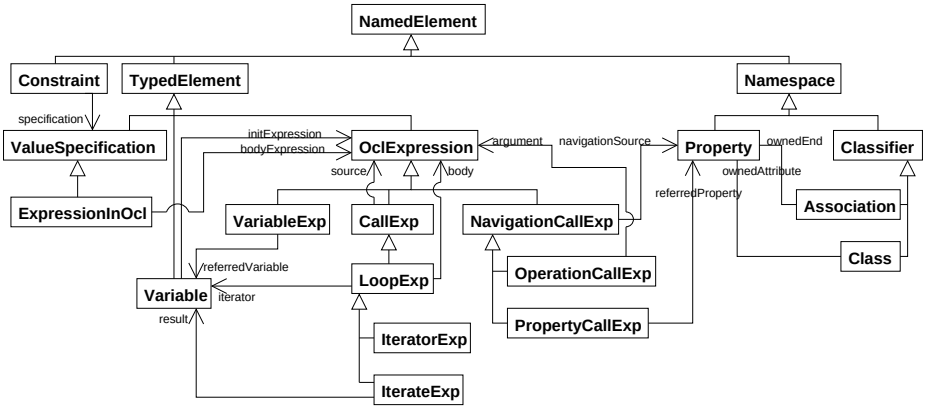


Fig. 4. Relevant excerpt of the combined UML and OCL meta-model

At first, we started to define the Many2One transformation using visual transformation rules. However, these rules quickly become very complex and hard to maintain. On this account, we defined some extensions to significantly reduce the complexity of the specifications, i.e. to reduce the number of transformation steps. In the following these extensions will be explained considering some Many2One transformation steps as example.

Step `existsToIterate`: If we take the example of Fig. 3 as a basis, the first transformation step which could be applied would be `existsToIterate`. In this example the expected result of this step would be the following expression:

```
self.employer->iterate( x; result:Boolean = false |
     result or x.location = self.location )
```

The transformation schema of the existsToIterate is depicted in the extended object diagram of Fig. 5.

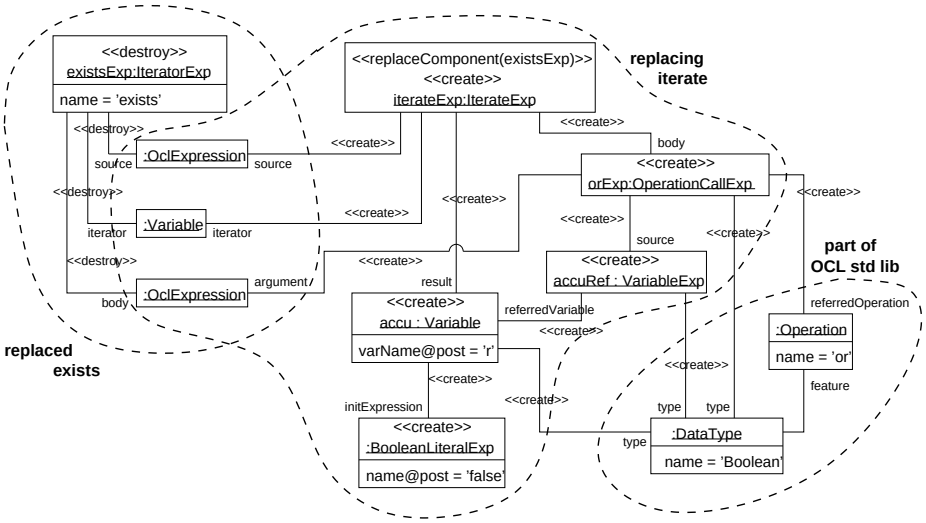


Fig. 5. Transformation step of `existsToIterate`

As mentioned in Section 2, all objects which are not created during this transformation step and their specified attribute values form the redex of this step. Within `existsToIterate` this redex consists of an exists expression (labeled with `replaced exists`) and the `or` operation and boolean type from the OCL standard library (marked as `part from OCL std lib`).

As explained before, the `existsToIterate` replaces the `exists` expression with an equivalent `iterate` expression. Consequently, the `replaced exists` part drops out during this step and is replaced by an iterate expression (labeled with `replacing iterate`) which holds the same source expression and control variables. By definition the iterate expression also owns an accumulator variable which is initialized with `false`. The body expression of `exists` now becomes the body expression of `iterate`, but is extended by the expression `result or`.

The replacement of expressions is one of our extensions which was introduced to reduce the number of transformation steps to a minimum. The replacement is realized by the stereotype «replaceComponent» and indicates that the owner of the labeled component should replace its part, specified by the parameter expression, with the caller component (e.g. in this case `exists` should be replaced with `iterate`). The effect is that every step only has to be defined once and not

for each possible owner (e.g. if, let, invariant and operation expressions) of the replaced expression. For this purpose we have introduced an ownership and a part relationship within the UML and OCL meta-model which are derived from existing associations.

The `@post` term indicates another extension within the transformation steps which describes the setting or changing of attribute values during the transformation step. An example of this construct can be found in Fig. 5 where the initialization value of the accumulator is set to `false` during the transformation.

Step `inlineRangeForRangeVar`: The `existsToIterate` step already needed some of the extensions to be realized. It has also demonstrated how transformation steps are specified within USE in general. The following transformation step (`inlineRangeForRangeVar`) will employ the last remaining extension. As mentioned above `inlineRangeForRangeVar` is one of three steps which realizes the transformation from `iterate` to an object-valued expression. It substitutes the source expression of `iterate` for all occurrences of the control variables within the body expression. Fig. 6 shows the transformation scheme of `inlineRangeForRangeVar`.

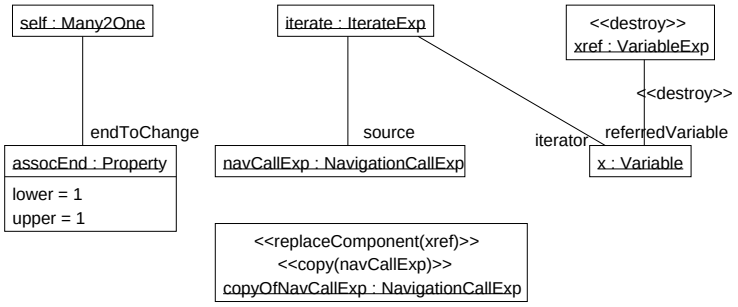


Fig. 6. Transformation step of `inlineRangeForRangeVar`

As before, the replacement of the control variable is realized using the «replaceComponent» stereotype. However, the new extension within this transformation step is the copying of an OCL expression. Within the UML and OCL meta-model several compositions, i.e. part-of relationships, are defined. These compositions state that an object should belong to at most one owner object. In our example transformation, the control variable can occur several times within the body expression of `iterate`. Because it should be substituted with the original source expression and because there can occur some part-of relationships to its owner, the source expression has to be copied.

This copying is visualized by the stereotype «copy» and is a mixture of shallow and deep copy. If the expression which should be copied has some composition relationships to its parts, these parts should be copied deeply. Otherwise, a

shallow copy (i.e. no copying of the related objects, but insertion of links between the new copy and these objects) is sufficient. The shallow copy also reduces the complexity of this approach.

As shown in this section, the defined extensions highly reduce the number of the transformation steps, especially the «replaceComponent» stereotype. We have also detected that some kind of copying is essential for the definition of more complex transformations.

3.3 Excerpt of the resulting USE specification

Finally, the following subsection shows an excerpt of the extended USE specification that is created from the last section's transformation specification by our translation filter. As explained in Sect. 2.3, one _redex() and one _apply() operations are created in the added transformation class (Many2One). We pick the two operations that are created for the step *inlineRangeForRangeVar* because they are short enough and yet implement several of the features that can occur in a step. Following the order of execution, we first show the _redex() operation, which tries to find a valid tuple of objects for inlineRangeForRangeVar (slightly reformatted to improve readability).

```
 1  Many2One::inlineRangeForRangeVar_redex() :
      Tuple(assocEnd:Property, navCallExp:NavigationCallExp,
            iterate:IterateExp, x:Variable, xref:VariableExp)
 2  begin
 3    declare foundRedex : Boolean
 4    set foundRedex := false
 5    for iterate : IterateExp in IterateExp.allInstances do
 6      for xref : VariableExp in VariableExp.allInstances do
 7        if (not foundRedex) and
 8           xref.referredVariable = iterate.iterator and
 9           Set{self, self.endToChange, iterate.source,
                 iterate, iterate.iterator, xref}->size = 6 and
10           assocEnd.lower = '1' and assocEnd.upper = '1'
11        then
12          set result := Tuple{assocEnd = self.assocEnd,
13                              navCallExp = iterate.source
                                  .oclAsType(NavigationCallExp),
14                              iterate = iterate,
15                              x = iterate.iterator,
16                              xref = xref}
17          set foundRedex := true
18        endif
19      next
20    next
21  end
```

This operation is basically a nested iteration over the potential matches for the step's context objects (lines 5 and 6). As an runtime optimization, the filter program tries to omit iterations for objects that implicitly reachable from other

redex objects via to-1 navigations. For example, *navCallExp* is reachable as *iterate.source*. The heart of the loops checks for each combination of candidates if the non-implicit required links exist between them (line 8). If further all objects are pairwise different and not undefined (line 9) and all other preconditions hold (line 10), the found redex is returned from the operation.

Having a valid redex, we can invoke the _apply operation, which looks as follows for our step:

```
1  Many2One::inlineRangeForRangeVar_apply(redex :
     Tuple(assocEnd:Property, navCallExp:NavigationCallExp,
           iterate:IterateExp, x:Variable, xref:VariableExp))
2  begin
3      declare copyOfNavCallExp : NavigationCallExp
4      set copyOfNavCallExp := redex.navCallExp.copy()
         .oclAsType(NavigationCallExp)
5      redex.xref.owner().oclAsType(ModelElement)
         .replace(redex.xref, copyOfNavCallExp)
6      delete (redex.xref, redex.x) from VariableExp_referredVariable
7      destroy redex.xref
8  end
```

This operation actually applies the step. Beside the basic step semantics (creating/destroying links/objects, setting attribute values, cf. [BG06]), this step operation also realizes the «copy» and «replaceComponent» extensions (lines 4 and 5). Both extensions require additional operations to be generated into the UML/OCL meta-classes (copy() and owner()). Both additional operations can be generated automatically by exploiting the compositions (the black diamonds) in the meta-model.

There is a lot more to say about the translation of our transformation language into .use files which does not fit into this paper (for example, the translation of the control condition). However, we hope that we have illustrated the general idea of using an imperative OCL as an assembler for our transformation language.

4 Conclusion

In the previous sections we showed how we are realizing UML model transformations with USE. We extended the USE tool itself and added a new imperative language component. We then translated the higher level transformation language that we used to implement our catalog onto this imperative language. We feel that several transformation languages or formalisms can be translated this way. Actually, we had to extend our initial transformation language by new elements («copy» and «replaceComponent») which underpins the flexibility of our approach.

Several alternatives exist for the imperative language extensions that we made. We believe that UML Actions (as part of the specification) could be used instead – actually, UML Actions provide a lot more features than we require

for our approach. The Kermeta meta-modelling environment could also be used for this purpose, as it supports OCL evaluation. Kermeta also provides several programming language-like extensions such as exception handling.

We do not aim to provide a "better QVT". Instead, our approach aims towards a support for evaluation and development of both, transformations, and transformation languages. Because USE (technically) and OCL (conceptually) can be reused, new transformations concepts can be developed hands-on.

References

[BG06] Fabian Büttner and Martin Gogolla. Realizing Graph Transformations by Pre- and Postconditions and Command Sequences. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Gregorz Rozenberg, editors, *Proc. 3rd Int. Conf. Graph Transformations (ICGT'2006)*, pages 398–412. LNCS 4178, Springer, Berlin, 2006.

[Büt05] Fabian Büttner. Transformation-Based Structure Model Evolution. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS'2005 Conference*, pages 339–340. Springer, Berlin, LNCS 3844, 2005.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Report of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. 2003.

[FDVF06] Franck Fleurey, Zoé Drey, Didier Vojtiseka, and Cyril Faucher. Kermeta language reference manual, 2006. http://www.kermeta.org/docs/KerMeta-Manual.pdf.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, August 1999.

[FUJ] The Fujaba tool suite, http://wwwcs.uni-paderborn.de/cs/fujaba/.

[GR02] Martin Gogolla and Mark Richters. Development of UML Descriptions with USE. In Hassan Shafazand and A Min Tjoa, editors, *Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002)*, pages 228–238. Springer, Berlin, LNCS 2510, 2002.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[MDFH04] Pierre-Alain Muller, Cédric Dumoulin, Frédéric Fondement, and Michel Hassenforder. The topmodL initiative. In *3rd Workshop in Software Model Engineering (WISME UML 2004), 11 October 2004, Lisbon, Portugal, Proceedings of the UML Satellite Activities 2004, Lecture Notes in Computer Science, Volume 3297, Feb 2005*, pages 242–245, 2004.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[Mod] ModelWare, http://www.modelware-ist.org/.

[OMG06] OMG. MOF QVT final adopted specification, 2006.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

Model-Driven Constraint Engineering

Michael Wahler[1], Jana Koehler[1], and Achim D. Brucker[2]

[1] IBM Zurich Research Laboratory, Saeumerstrasse 4, 8803 Rueschlikon, Switzerland
[wah,koe]@zurich.ibm.com
[2] Information Security, ETH Zurich, 8092 Zurich, Switzerland
brucker@inf.ethz.ch

Abstract. A high level of detail and well-formedness of models have become crucial ingredients in model-driven development. Constraints play a central role in model precision and validity. However, the task of constraint development is time-consuming and error-prone because constraints can be arbitrarily complex in real-world models.
To overcome this problem, we propose a solution that we call *model-driven constraint engineering*. In our solution, we define the notion of computation-independent constraints that are provided in the form of meta-model integrated patterns. The parameterized patterns are transformed into platform-independent or platform-independent constraints by a model transformation. In addition, we show how our approach can be supported by a tool.

1 Introduction

In model-driven engineering, textual constraints are used to express details about a model that are either hard or even impossible to express in a diagrammatic way. For instance, hundreds of constraints are used in the specification of the UML (Unified Modeling Language [24]) meta-model. Constraints stem from different sources: there may be legal restrictions that a system needs to obey; there may be company policies that grant privileges to certain kinds of customers; there may be technical restrictions on a system [8]; there may be security restrictions [20]; and there may be facts that are implied by common sense that cannot be expressed diagrammatically.

While models were solely used for documentation and communication purposes in the past, recent model-centric development approaches such as MDA (Model Driven Architecture [19]) use models as first-class artifacts in the development process. For instance, business process models can be transformed to executable code that is run on process execution engines [17] or models in a domain-specific security language are transformed to UML [7]. To guarantee the correctness of the execution of the generated code, it is crucial that every model instance conforms to its defining model and satisfies its constraints. These validity checks can be performed automatically if the constraints are formalized. For instance, tools exist that type-check a set of OCL (Object Constraint Language [23]) constraints and validate a model against them [3]. Alternatively,

validity checks can be implemented in a programming language, e.g., Java, using a model access API, e.g., EMF (Eclipse Modeling Framework [14]).

The creation and maintenance of constraints is a tedious task. In a case study in the business modeling environment that we performed, about 80 constraints were necessary to guarantee the executability of a behavioral model for business process monitoring. All the constraints are invariants on the model elements and restrict the set of allowed model instances to a set that is executable on a process execution engine. While some of these constraints were rather simple, many complex constraints needed to be formalized, which turned out to be a time-consuming and error-prone task. The formalization resulted in approximately 500 lines of OCL code, which by nature are unlikely to be bug-free.

Even when the constraint expressions or validation code do not contain any errors, they need to be adapted once the model changes. This usually results in additional time-consuming coding and debugging phases, especially in model refactorings [11,21] where models undergo frequent changes and the attached constraints need to be kept consistent with new versions of the model.

Our contribution to solving the problem of constraint development consists of three parts. Firstly, we introduce the notion of computation-independent constraints and transformations to platform-independent constraints. Secondly, we introduce constraint patterns and separate the patterns into atomic and composite patterns and add a structure to them to enhance their expressiveness and usability. Thirdly, we discuss the requirements for tool support and illustrate our prototype for Eclipse/UML2 [13].

We believe that a flexible pattern-based approach that is supported by a tool offers an important improvement for constraint engineering. Most syntactic and semantic errors can be avoided because the developer can generate OCL code instead of writing it by hand. Furthermore, our solution promises to decrease development time substantially.

2 Background

Our solution is based on the idea of constraint patterns (sometimes called *idioms*) for UML models [1,2]. A constraint pattern is a parameterized formula that can be instantiated to a constraint by providing values for its parameters. In [1], only two structural constraint patterns—*Semantic Key Attribute* and *Invariant for an Attribute Value of a Class*—are presented, which we consider too little to have a relevant impact on solving the aforementioned problem.

The semantics of a constraint pattern can be provided in any language, e.g., parameterized OCL templates such as in [1]. This has the advantage that an OCL constraint can be simply instantiated by providing values for the pattern parameters. In our solution that we call *model-driven constraint engineering* we follow the MDA approach [19] that comprises models at different levels of abstraction. We consider a constraint pattern a computation-independent model (CIM) of a constraint. A CIM constraint can be transformed into a platform-independent or platform-specific model (PIM/PSM) by a model transformation.

CIM constraints are integrated into the UML meta-model. This integration is accomplished by using the meta-model representation of model elements as parameters for a constraint instead of their textual representation. The types of these parameters are thus elements from the UML meta-model and can be both object types, e.g., *Property*, or simple types, e.g., *String*. The PIM constraint, namely the constraint in a concrete syntax of the constraint language—in our case, OCL—is automatically generated from model-integrated constraint using a model transformation.

We illustrate these concepts in Fig. 1. On the left hand side of this figure, a class *Employee* is shown. This class owns a property whose name is *name* and whose type is *String*. Class *Employee* is constrained by *C1*, which has one parameter, *targetAttribute*, which is an association to a UML *Property*. Furthermore, this constraint is stereotyped *UniqueAttributeValue* which denotes a constraint pattern. Thus, *C1* is the CIM of a constraint.

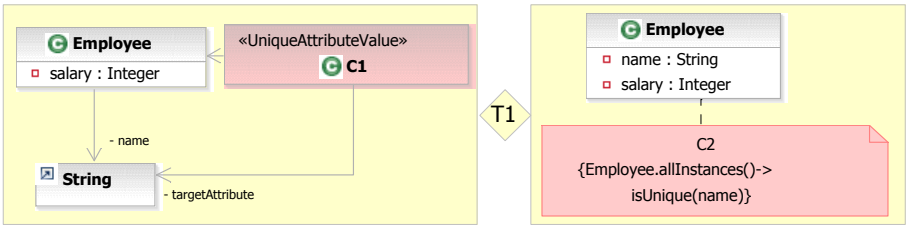


Fig. 1: Transformation from CIM to PIM

The transformation *T1* provides the semantics for the CIM *C1* which is given in this case as a parameterized OCL template. The result can be seen on the right hand side of Fig. 1 where *C2* is the result from transforming *C1* into a platform-independent constraint. By replacing the transformation *T1*, different target platforms can be served. For instance, instead of generating an OCL expression, Java code could be generated that implements a constraint in the Eclipse/EMF [14] framework. In this case, platform-specific knowledge has to be provided in the transformations because our constraint patterns are computation- and platform-independent.

3 Example Model and Constraints

In Fig. 2 we illustrate a simple model that serves as running example. The UML class diagram contains two classes, *Manager* and *Employee*. These classes are related by a many-to-many relation. An instance of *Employee worksFor* at least one manager; a manager *employs* any natural number of employees.

Besides the defined classes and associations, instances of this model are not restricted in any way. There may be managers without employees, and employees may have a salary of zero but work for multiple managers.

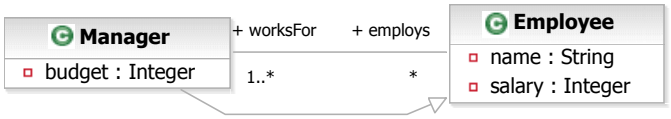


Fig. 2: Manager and Employee Class Diagram

We assume fictitious labor union and company IT requirements that every work environment has to satisfy. The requirements are captured in the following constraints informally in English and formally as OCL expressions.

Constraint 1. *A manager must employ at least one employee with a salary of at least 3000.*

This constraint requires that for each instance m of *Manager* there exists an instance e of *Employee* that is related to m by the relation *employs*. Furthermore, the value of the *salary* attribute of e must be at least 3000.

```
context Manager
inv: self.employs->exists( e | e.salary >= 3000)
```

Constraint 2. *A manager may not occur within his management hierarchy.*

This constraint prevents that a manager m is responsible for him-/herself by being related to him-/herself directly by the *worksFor* relation or indirectly by other managers $\{m_i, \ldots, m_j\}$ that work for m. For the corresponding OCL expression, we need to define an operation closureWorksFor(S) that computes the transitive closure [4] of the *worksFor* relation. A parameter S in which elements already processed are stored ensures the termination of this operation.

```
context Manager
def: closureWorksFor(S:Set(Manager)) : Set(Manager) =
     worksFor->union((worksFor - S)->
       collect(m : Manager | m.closureWorksFor(S->including(self)))->asSet())
inv: not self.closureWorksFor(Set{})->includes(self)
```

Constraint 3. *The company may not have more than five organizational layers.*

This constraint restricts the depth of the *worksFor* navigation path. Because a manager can employ another manager, arbitrary hierarchy levels can be realized. However, the fictitious labor union forbids more than five hierarchy levels. A recursive query pathDepth() needs to be defined to compute the path depth. This query has two parameters, *max* and *counter*, where *max* is set to the desired maximum path depth minus 1 and *counter* is initialized with 0.

```
context Manager
def: pathDepth(max:Integer, counter:Integer): Boolean =
     if (counter > max or counter < 0 or max < 0) then false
     else if (self.worksFor->isEmpty()) then true
          else self.worksFor->forAll(m:Manager|m.pathDepth(max, counter+1))
          endif
     endif
inv: self.pathDepth(4,0)
```

Constraint patterns can be identified by analyzing existing constraints and abstracting from them. For example, the constraints introduced in this section can be generalized to the following expressions. Constraint 1 requires the existence of an instance that is related to the context object and has a value restriction on an attribute. Constraint 2 prevents cyclic navigation paths in a model instance. Constraint 3 can be generally seen as a constraint that restricts the maximum length of a navigation path.

From these general expressions, constraint patterns can be derived and described using a schema similar to the one described in [1]. We call the pattern used for Constraint 1 *Exists*, the pattern derived from Constraint 2 *CyclicDependency* and the pattern from Constraint 3 *PathDepthRestriction*. These patterns are part of the taxonomy that we introduce in the following section.

4 A Taxonomy of Structured, Computation-Independent Constraint Patterns

Although the constraint pattern approach as introduced in [1] reduces both the development time and error rate for model constraints, it has one important restriction. As each pattern represents a subset of all possible constraint expressions, even with an extensive pattern library, there will be many constraints that are not expressible in terms of existing constraint patterns.

Therefore, we introduce the notion of *structured constraint patterns* that add a high degree of expressiveness to the existing constraint pattern approach by two measures. Firstly, we divide constraint patterns into *atomic* and *composite* patterns where we introduce a large set of atomic patterns. Composite patterns are recursively constructed from atomic patterns. Secondly, we introduce the logical concepts of implication and negation that allow the applicability of an instance of a constraint pattern to be restricted.

4.1 Atomic Constraint Patterns

In this section we present an extensible library of atomic constraint patterns. The constraint patterns are related with generalization associations. Therefore, we create a *taxonomy* of patterns. The taxonomy gives a structure to the set of patterns and helps one to find the right pattern for a specific purpose.

The idea of atomic constraint patterns is to identify a large set of atomic constraints that restrict fundamental concepts of a model, e.g., attribute values or relations between objects. Furthermore, atomic constraints can be referenced from composite constraints to create a complex constraint from several components. The atomic constraint patterns that we have identified are illustrated in Fig. 3 where we included the two structural patterns from [1] as *UniqueAttributeValue* and *AttributeValueRestriction*. The patterns refer to the UML meta-classes *Class* and *Property* and to the OCL meta-class *OclExpression*.

In MDA, the semantics of a model is inherent in the model transformations that generate PIM constraints from parameterized CIM constraint patterns. In

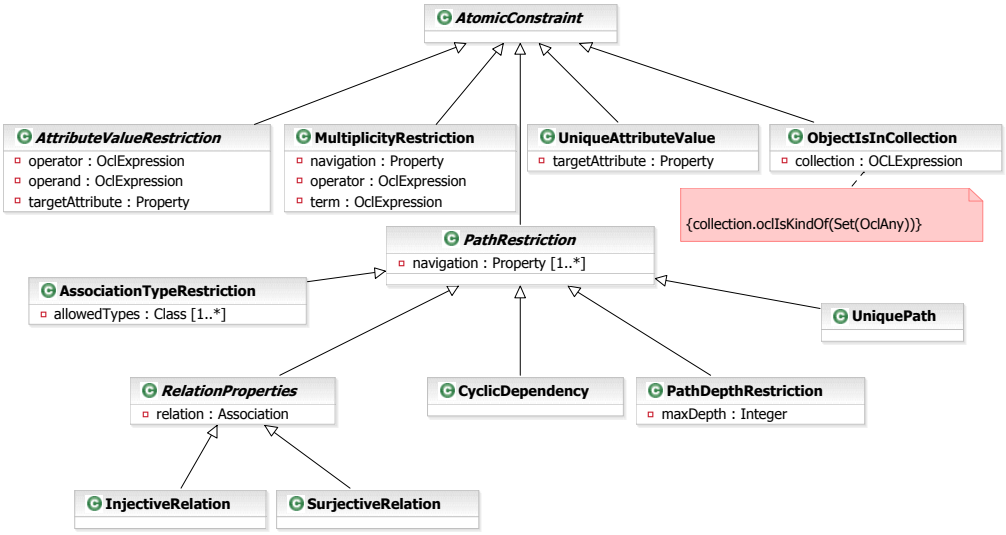


Fig. 3: UML Class Diagram of Atomic Constraint Patterns

the following, we provide informal semantics for the patterns in Fig. 3 and discuss the model transformations in Sect. 5.

Pattern descriptions. The *MultiplicityRestriction* pattern restricts the multiplicity of an association. Although the multiplicity of an association can be restricted in a UML class diagram, this pattern allows for multiplicity restrictions that depend on properties of the model instance, e.g., an attribute value.

Two constraint patterns target at attribute values. The *AttributeValueRestriction* can be used to restrict the value of an attribute of a class for all instances of the class. The *UniqueAttributeValue* pattern requires that all instances of the constrained class have distinct values for the specified target attribute.

The *ObjectIsInCollection* pattern can be used to require that the context element is in the specified collection of objects. For instance, we could require that each manager is in the set of his employees in Fig. 2.

In the lower part of Fig. 3, we show patterns that can be generalized to *PathRestriction* constraints. These patterns restrict properties of a navigation path in a model instance. The *AssociationTypeRestriction* pattern can be used to restrict an association a that is defined on a general class C_0 in a way that in an instance, only certain subclasses $C_1, \ldots, C_n$ of C_0 may participate in the relation that is defined by a.

The *CyclicDependency* pattern can be used to require cycles in the instance graph of the model. Such a cycle can occur if an instance element is related to itself with a certain navigation. This navigation is the only parameter for this constraint pattern. An example for an instance of this pattern is Constraint 2.

The *InjectiveRelation* and *SurjectiveRelation* patterns can be used to establish the mathematical concepts of injective ($f(a) = f(b) \rightarrow a = b$) and surjective ($f : X \rightarrow Y \wedge \text{range}(f) = Y$) relations. Bijective relations can be modeled by constraining an element with a constraint for injectivity and one for surjectivity.

The *UniquePath* pattern can be used to constrain that there may not be more than one path from the context element to a related element. An infamous configuration that can be excluded with this pattern is the "diamond of death" in object-oriented programming languages [22].

The *PathDepthRestriction* pattern can be used to restrict the maximum path length in a model instance for reflexive associations. Constraint 3 from our example is an instance of this pattern where the maximum length of the *employs* association to 5.

4.2 Composite Constraint Patterns

Apart from atomic constraint patterns, which each represent one property of a model element, composite constraints can be used to express complex properties of a model. Therefore, they can integrate an arbitrary number of other constraints (either atomic or composite). Thus, complex constraints can be developed by combining several simple constraints.

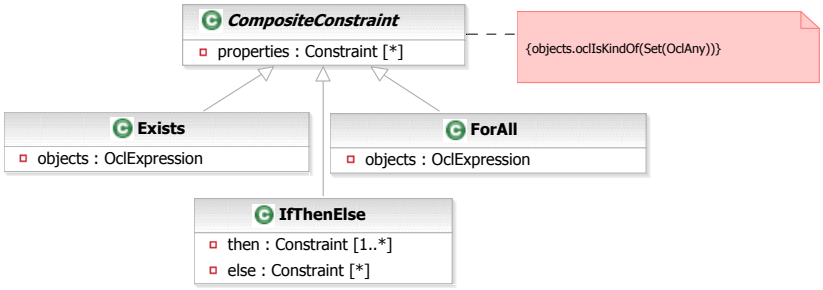


Fig. 4: Class Diagram of Composite Constraint Patterns

So far, we have identified three composite constraint patterns, *Exists*, *ForAll* and *IfThenElse*. Constraint 1 is an example instance of the *Exists* pattern: for the context element m of class *Manager* there has to exist an element e that is related to m with the navigation *employs*. This element e needs to satisfy a number of constraints, the *properties* of the composite constraint. The *ForAll* constraint pattern is similar except that *all* elements in the object collection need to satisfy the properties specified.

The *IfThenElse* pattern realizes an if-then-else expression. If the context element of the constraint satisfies all *properties*, it also needs to satisfy all *then* constraints, otherwise, it needs to satisfy all *else* constraints.

4.3 Adding Logical Structure to Constraint Patterns

We add structure to the concept of constraint patterns by introducing the concepts of negation and implication in a class *StructuredConstraint*, which is a specialization of the UML meta-class *Constraint*. Class *StructuredConstraint* has

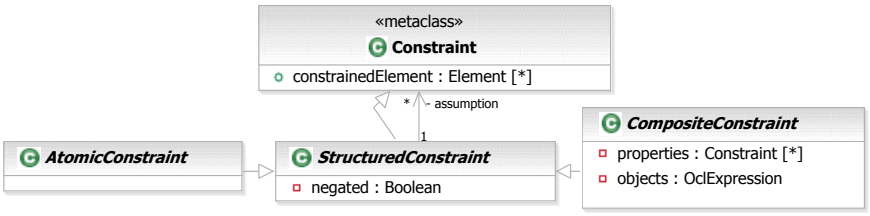


Fig. 5: UML Class Diagram Overview of Structured Constraint Classes

two child classes, namely the previously introduced classes *AtomicConstraint* and *CompositeConstraint*. This idea is illustrated in Fig. 5.

The concept of logical implication is realized as follows. Each structured constraint c can have a finite set A of assumptions that can be any kind of constraint. This is illustrated by the association *assumption* from *StructuredConstraint* to *Constraint*. This allows us to use either arbitrary constraints (defined by a UML *ValueSpecification*) or structured pattern instances as assumptions for constraints. The semantics of the *assumption* relation is defined as follows: Let c be an instance of a structured constraint and A be a finite set of constraints that is related to c with the *assumption* relation. Then the conjunction of all constraints in A implies c. The concept of logical negation is realized by the attribute *negated* of the class *StructuredConstraint*.

5 Transforming CIM to PIM

Having defined a library of CIM constraint patterns, we provide the transformation definitions that are necessary to generate PIM constraints from the parameterized patterns. In this section, we introduce a transformation that generates OCL constraints from parameterized CIM constraint patterns. The transformation transform_OCL(c) uses OCL templates to generate output. We use pseudo code that has the same expressivity as common programming languages for the definition of the operations.

Three steps are necessary to transform an atomic constraint pattern. First, the assumptions need to be generated. Therefore, we define a function transform_assumptions_OCL(c) (Listing 1.1). Then, the OCL keyword not is inserted if the pattern attribute *negated* is *true*. Finally, the variables in the templates for the constraint patterns are replaced by concrete values from the pattern specification. The OCL templates are shown in Table 1.

```
sub transform_assumptions_OCL( c: StructuredConstraint ) {
  # print the conjunction of assumptions
  foreach p in c.assumption
     print (transform_OCL(p) + " and ");

  # print the implication operator;
  # it needs to be preceded by 'true' to generate correct syntax
  print "true implies ";
}
```

Listing 1.1: Transformation Function for assumptions

For conciseness, we do not present a definition of the replace_parameters(t) function. Listing 1.2 shows complete the transformation from CIM to PIM for an atomic pattern.

```
sub transform_OCL(c:AtomicConstraint) {
  # print the assumptions of the constraint
  transform_assumptions_OCL(c);

  # print the OCL keyword 'not' if the constraint is negated
  if (c.negated) print "not ";

  # replace the variables in the template and print constraint
  print replace_parameters(template(c));
}
```

Listing 1.2: OCL Transformation Function for Path Depth Restriction Pattern

Pattern Name	**Template**
PathDepthRestriction	**def**: pathDepth(max:Integer, counter:Integer): Boolean = if (counter > max or counter < 0 or max < 0) then false else if (self.<*navigation*>−>isEmpty()) then true else self.<*navigation*>.forAll(e|e.pathDepth(max,counter+1)) endif endif **inv**: self.pathDepth(<maxDepth>−1,0)
MultiplicityRestriction	**inv**: self.<*navigation*>−>size() <*operator*> <*term*>
AttributeValue–Restriction	**inv**: self.<*targetAttribute*> <*operator*> <*operand*>
UniqueAttributeValue	**inv**: self.allInstances()−>isUnique(<*targetAttribute*>)
ObjectIsInCollection	**inv**: self.<*navigation*>−>includes(self)
AssociationType–Restriction	**inv**: self.<*navigation*>−>forAll(x | <*allowedTypes*> −>exists(c | x.oclIsTypeOf(c)))
CyclicDependency	**def**: closure<*navigation*>(S:Set(OclAny)) : Set(OclAny) = <*navigation*>−>union((<*navigation*> − S). closure<*navigation*>(S−>union(<*navigation*>))−>asSet()) **inv**: not self.closure<*navigation*>(Set{})−>includes(self)
InjectiveRelation	**inv**: self.<*navigation*>.lower = 1 and self.<*navigation*>.upper = 1 and self.allInstances()−>forAll(x1,x2 | x1.<*navigation*> = x2.<*navigation*> implies x1=x2)
SurjectiveRelation	**inv**: self.<*navigation*>.allInstances()−>forAll(y | y.<*relation*>−>size() >= 1)
UniquePath	**inv**: self.<*navigation*>−>forAll(x | self.<*navigation*>−>count(x)=1)

Table 1: OCL Templates for Atomic Constraint Patterns

The composite constraints we introduced use other constraints as *properties* for the elements in their object collections. This higher-order use of constraints makes the code generation slightly more complicated than for atomic constraints. A special transformation needs to be written for each composite pattern. For instance, a transformation function for the *Exists* pattern is shown in Listing 1.3. The transformation function for the *ForAll* is similar; only line 6 needs to be adapted. The *IfThenElse* pattern can be transformed analogously.

The OCL generation works as follows. First, the assumptions and the negation are generated if necessary (lines 2,3). Then, the header for the existential quantification over the object collection is generated (line 6) where e is the variable that represents an element of the collection. In lines 9-12, a conjunction of expressions is created from the *properties* of the *Exists* pattern. In this conjunction, every occurrence of the keyword *self* is replaced by the bound variable e. Finally, the conjunction is concluded with the constant *true* and a closing bracket is added (line 15).

```
1  sub transform_OCL( c : Exists ) {
2    transform_assumptions_OCL(c);
3    if (c.negated) print "not ";
4
5    # print the first part of the constraint body and open bracket
6    print "self."+c.objects+"→exists( e | ";
7
8    # print the properties that e needs to satisfy
9    foreach p in c.properties {
10         print transform_OCL(c.properties).replace("self", "e");
11         print "and ";
12   }
13
14   # print a finalizing 'true' and closing bracket
15   print "true )";  }
```

Listing 1.3: Transformation Functions for Composite Constraints

6 Tool Support for Model-Driven Constraint Engineering

Tool support is essential for the acceptance and success of model-driven engineering approaches. In the following, we present how we employ our idea of structured CIM constraint patterns in a model-driven development tool.

As depicted in Fig. 5, our concept of structured constraint is a specialization of the UML meta-class *Constraint*. We suggest an implementation of our approach as UML Profile where each structured constraint pattern is represented by a UML stereotype. The taxonomy of constraint patterns is realized using generalization associations between the stereotypes. The attributes of the constraint patterns become attributes of the stereotypes in the implementation. Here, one deficiency of UML 2.0 becomes critical. In UML 2.0, stereotypes may not have associations with meta-classes [24]. Thus, a *UniqueAttributeValue* constraint cannot refer to the UML meta-class *Property*. Even worse, a composite constraint cannot refer to other constraints that elements need to satisfy. However, this deficiency no longer exists in UML 2.1 [25], where associations between a stereotype and a meta-class may be defined.

Fig. 6: Screenshot of Eclipse/UML Profile Editor

The Eclipse UML2 project [13] provides an implementation of the UML 2.1 meta-model based on the Eclipse Modeling Framework [14]. This makes Eclipse/UML2 an ideal platform for realizing tool support for structured constraint patterns. In Fig. 6 we show a screenshot of the UML Profile editor in Eclipse. As can be seen, the taxonomy of structured constraint patterns can be implemented in a straight-forward manner.

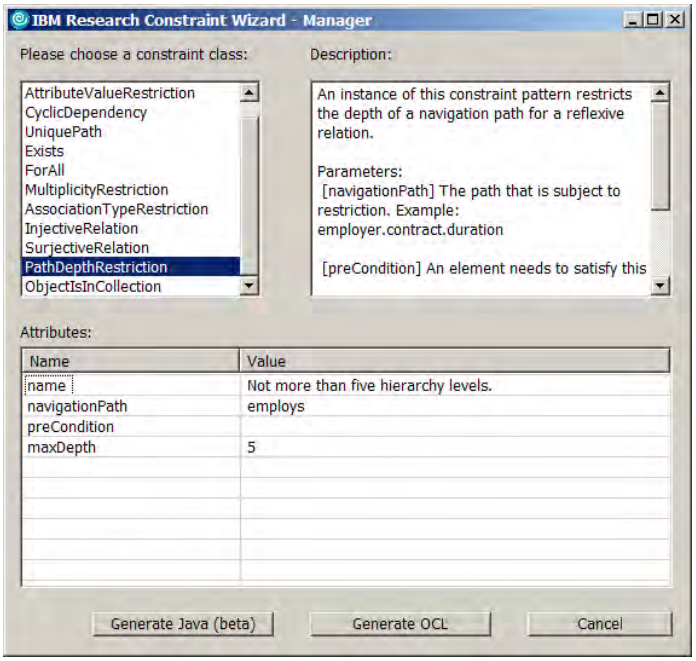


Fig. 7: Screenshot of Constraint Wizard

We prototyped a graphical user interface that guides a user during constraint creation and maintenance. In Fig. 7 we show a screenshot of our "wizard". In the top left window, the user can choose a constraint pattern. When a pattern is selected, a description of the pattern and its parameters are shown in the top right

part of the window. In the bottom part, the attributes of the selected pattern are shown and values can be entered for them. As can be seen, the wizard implements one CIM-to-PIM transformation that generates OCL expressions and one CIM-to-PSM transformation that creates Java code for run-time model validation. Furthermore, the wizard can also be used to modify previously created structured constraints.

6.1 Applying the Tool to the Example

We have claimed throughout this paper that our approach helps to decrease development time and rate of syntactic errors. To indicate the practicability of our approach, we revisit the example from Sect. 3 and apply our method to it. Using the constraint wizard prototype, we choose appropriate patterns for Constraints 1–3 and specify their parameters.

The result can be seen in Fig. 8. The class *Manager* is constrained by two atomic constraints. An instance of the *PathDepthRestriction* pattern, representing Constraint 3, and an instance of the *CyclicDependency* pattern representing Constraint 2 are attached to *Manager*. Constraint 1 is realized as an instance of the composite pattern *Exists*: among the set of all employees (*self.employs*), at least one element needs to satisfy the *AttributeValueRestriction* property that is attached to the composite constraint.

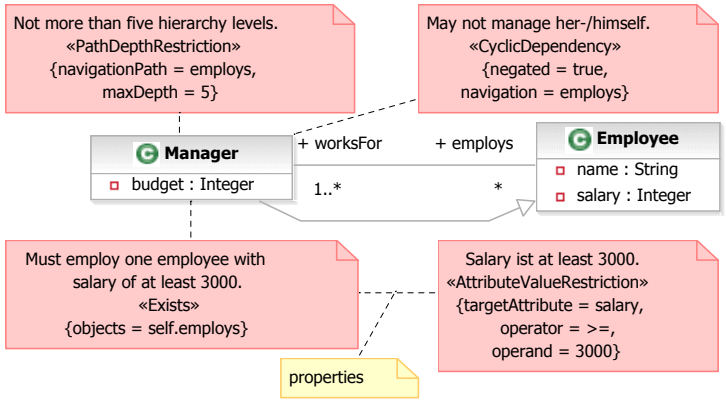


Fig. 8: Example Class Diagram with Structured Constraints Attached

This small example shows the benefits of our approach. The two complicated constraints, Constraint 2 and 3, can be specified by simply providing two parameter values each. If requirements change, these constraints can be quickly adapted without reading, adapting and testing verbose expressions. Constraint 1 is split into two parts, a quantification and a predicate part. This allows for advanced graphical input support that may help users without background in formal languages.

We believe that this small example already shows the practicability of our approach. Complicated recursive expressions are replaced by structured, concise

and easy-to-read constraint definitions. In addition, our model-driven approach enables the automatic generation of platform-independent or platform-specific constraints in various languages or modeling frameworks.

7 Related Work and Conclusion

The difficulty of developing concise and correct OCL constraints has been addressed in numerous publications. OCL is considered to be a very important formalism in today's modeling technologies, still unsolved problems make constraint development difficult [9]. Several solutions have been proposed for dealing with the complexity of constraints and syntactic hurdles. In [10], a set of recommendations is provided to increase correctness, clearness and efficiency of OCL specifications. To simplify the syntax of OCL, a visual concrete syntax for OCL is proposed in [6].

Several publications use the idea of constraint patterns, thus following up the general idea introduced in [15]. Patterns for model-driven development constraints were first mentioned in [5], where one pattern – *Singleton* – is introduced. The idea of constraint patterns is further elaborated on in [1,2], where a small number of constraint patterns are introduced along with OCL templates.

Here, we have introduced the idea of model-driven constraint engineering. Our approach goes beyond existing work in three directions. Firstly, we have introduced the notion of computation-independent patterns and transformations to platform-independent constraints. Secondly, we have introduced a library of patterns that goes far beyond existing pattern solutions in quantity and quality and provided a high degree of expressiveness to this approach by adding logical structure and classifying patterns into atomic and composite patterns. Thirdly, we provide tool support for applying the concepts in a real development environment.

We claim that our approach helps to decrease the time and error rate for constraint development. For instance, the OCL expression that is necessary to express Constraint 3 (Sect. 3) uses a recursive definition that is not easy to understand. In contrast to the lengthy and complicated OCL statement, the same constraint can be defined as an instance of the *PathDepthRestriction* pattern. Appropriate tool support (Fig. 7) further reduces the problem of defining a constraint by pointing-and-clicking to relevant model elements.

The patterns that we present in this paper were elicited from a large set of structural constraints for a model in the business process modeling domain. From the current constraint patterns, almost 90% of the constraints in our case study (cf. Sect. 1) can be instantiated. We believe that more interesting constraint patterns can be identified in other application domains, e.g., model transformations [18], ontology modeling [12] or model refactorings [16]. Therefore, we envision to make the taxonomy publicly available such that any interested user can use the approach and extend the constraint pattern library.

Future work includes the definition of new atomic and composite constraint patterns. However, a pattern-based approach such as the one that we introduce

in this paper is always a tightrope walk between simplicity and completeness with respect to the expressivity of the underlying constraint language. On the one hand, patterns are there to simplify the definition of constraints by providing abstractions for commonly used constraint expressions. On the other hand, given a set of constraint patterns, you can always find a constraint that cannot be expressed as an instance of the available patterns. Adding as many patterns in as much detail as possible to the taxonomy will eventually turn the taxonomy into a meta-model of the OCL language specification. Such a fine granularity would not help with the initial problems of time consumption and error rate.

As a rule of thumb, we discourage the introduction of trivial patterns for two reasons. Firstly, trivial patterns can always be replaced by short OCL expressions. Secondly, a large number of patterns makes it difficult to keep an overview and select the "right" pattern for a specific purpose. For this reason, we discourage the use of the *AttributeValueRestriction* pattern, which we included in this paper for "historic" reasons only. The other patterns will be subject to discussion whether they simplify matters or introduce additional overhead. We believe that further case studies can clarify this issue. Future work also includes the development of constraints for the patterns themselves that deal with problems such as meaningful input ranges or type safety for the values of pattern variables.

We would like to emphasize that although we have introduced a *wizard*, we cannot spirit away the complexity inherent to many constraints. However, we believe that that our approach offers a powerful tool for dealing with this inherent complexity.

Acknowledgements

We would like to thank David Basin, Jochen Küster, Alexander Pretschner and Ksenia Ryndina for their valuable feedback on earlier versions of this paper.

References

1. J. Ackermann. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. In T. Baar, editor, *Workshop on Tool Support for OCL and Related Formalisms*, Technical Report LGL-REPORT-2005-001, pages 15–29. EPFL, 2005.
2. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, 4(1):32–54, 2005.
3. J. A. T. Álvarez, V. Requena, and J. L. Fernández. Emerging OCL Tools. *Software and System Modeling*, 2(4):248–261, 2003.
4. T. Baar. The Definition of Transitive Closure with OCL – Limitations and Applications. In A. Ershov, editor, *Perspectives of System Informatics*, 2003.
5. T. Baar, R. Hähnle, T. Sattler, and P. H. Schmitt. Entwurfgesteuerte Erzeugung von OCL-Constraints. *Softwaretechnik-Trends*, 20(3), 2000.
6. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.

7. A. D. Brucker, J. Doser, and B. Wolff. A Model Transformation Semantics and Analysis Methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006*, number 4199 in LNCS, pages 306–320. Springer-Verlag, Genova, 2006.
8. S.-K. Chen, H. Lei, M. Wahler, H. Chang, K. Bhaskaran, and J. Frank. A model driven XML transformation framework for Business Performance Management model creation. In *International Journal of Electronic Business*, volume 4. Inderscience, 2006.
9. D. Chiorean, M. Bortes, and D. Corutiu. Proposals for a Widespread Use of OCL. In T. Baar, editor, *Workshop on Tool Support for OCL and Related Formalisms*, Technical Report LGL-REPORT-2005-001, pages 68–82. EPFL, 2005.
10. D. Chiorean, D. Corutiu, M. Bortes, and I. Chiorean. Good Practices for Creating Correct, Clear and Efficient OCL Specifications. In *NWUML 2004*, 2004.
11. A. L. Correa and C. M. L. Werner. Applying Refactoring Techniques to UML/OCL Models. In T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, editors, *UML*, volume 3273 of *LNCS*, pages 173–187. Springer, 2004.
12. S. Cranefield and M. Purvis. UML as an Ontology Modelling Language. In *IJCAI 99*, 1999.
13. The Eclipse UML2 Project. `http://www.eclipse.org/uml2/`.
14. The Eclipse Modeling Framework. `http://www.eclipse.org/emf`.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA, 1995.
16. M. Gogolla and M. Richters. Expressing UML Class Diagrams Properties with OCL. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 85–114, London, UK, 2002. Springer-Verlag.
17. R. Hauser and J. Koehler. Compiling Process Graphs into Executable Code. In *Third International Conference on Generative Programming and Component Engineering*, volume 3286 of LNCS, pages 317–336. Springer, 2004.
18. R. Hauser, J. Koehler, S. Sendall, and M. Wahler. Declarative Techniques for Model-Driven Business Process Integration. *IBM Systems Journal*, 44(1), 2005.
19. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
20. T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.
21. S. Markovic and T. Baar. Refactoring OCL Annotated UML Class Diagrams. In *MODELS 2005*, volume 3713 of *LNCS*, pages 280–294, 2005.
22. R. C. Martin. Java and C++. A Critical Comparison. Online document. `www.objectmentor.com/resources/articles/javacpp.pdf`, March 1997.
23. Object Management Group (OMG). UML 2.0 OCL Final Adopted Specification. `http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf`, 2003.
24. Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0. OMG Document formal/05-07-04, July 2005.
25. Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.1. OMG document ptc/06-04-02, April 2006.

OCL support in an industrial environment

Michael Altenhofen[1] and Thomas Hettel[2] and Dr. Stefan Kusterer[3]

[1] SAP Research, CEC Karlsruhe, 76131 Karlsruhe, Germany,
michael.altenhofen@sap.com
[2] SAP Research, CEC Brisbane, Brisbane, Australia
thomas.hettel@sap.com
[3] SAP AG, 69190 Walldorf, Germany
stefan.kusterer@sap.com

Abstract. In this paper, we report on our experiences integrating OCL evaluation support in an industrial-strength (meta-)modeling infrastructure. We focus on the approach taken to improve efficiency through what we call *impact analysis* of model changes to decrease the number of necessary (re-)evaluations. We show how requirements derived from application scenarios have led to design decisions that depart from or resp. extend solutions found in (academic) literature.

1 Introduction

The MDA [1] vision describes a framework for designing software systems in a platform-neutral manner and builds on a number of standards developed by the OMG. Within this paper, we are mostly concerned with the Meta Object Facility (MOF) [2] used to define, manipulate and integrate meta-data and data in a uniform manner, and the Object Constraint Language (OCL) [3], originally designed as an extension to the Unified Modeling Language (UML) [4] to formally capture additional integrity constraints on UML models. In the meantime, the scope of OCL has been extended towards a model query language suitable for supporting model-to-model transformation tasks [5].

With its upcoming standard-compliant modeling infrastructure, SAP plans to support large-scale MDA scenarios with a multitude of meta-models that put additional requirements on the technical solution that are normally considered out-of-scope in academic environments. This may lead to solutions that may be considered inferior at first sight, but actually result from a broader set of (sometimes non-functional) requirements.

This paper focuses on one particular aspect in SAP's modeling infrastructure, namely a efficient support for OCL, both as a constraint language for meta-models and as query language for other tasks, such as model-to-model transformations or generic tool support. We will show how we have modified some of the existing approaches to better fit the requirements we're facing in our application scenarios.

The rest of the paper is organized as follows: In section 2, we will give an overview of SAP's modeling infrastructure focusing on features that are considered critical in large-scale industrial environments. Then, in section 3, we will

summarize related work in the area of OCL impact analysis that has guided our work leading to a more detailed description of our approach in section 4. In section 5, we will report on first experimental experiences and conclude in section 6 by summarizing our work.

2 The SAP Modeling Infrastructure (MOIN)

Mid of 2005, SAP launched "Modeling Infrastructure" (MOIN), a development project within the NetWeaver[4] organization. The goal of the MOIN project is to implement the platform for SAP's next generation of modeling tools.

2.1 Modeling at SAP

In the past years, SAP's development teams made extensive use of modeling approaches, convinced that this shortens development cycles and improves quality of both, software designs and implementations. Especially in the area of business process platform development, modeling has proven as highly efficient.

Often without being aware of it, various teams advanced existing design-time tools into special purpose modeling solutions. In most cases, being very powerful with respect to the intended use case, these tools show deficiencies concerning interoperability. One becomes aware of this after looking at the complete suite of tools, which has developed over time.

A key to consolidation of the tool suite lies in establishing one common platform, which offers all the services required by these tools.

2.2 Overview on the Architecture and Services of MOIN

The requirements for MOIN resulted in an architecture, which consists of the components described in the following sections as major building blocks.

Repository First and foremost, MOIN is a repository infrastructure for meta-models and models. In particular it is capable of storing any MOF compliant meta-model together with all the associated models. For accessing this content, JMI compliant interfaces can be used, which are generated for the specific meta-model. This allows client applications to navigate and manipulate both, models and meta-models.

The MOF standard does not impose any concepts for physical structuring of model content onto the implementer. However, for features like locking or even the simplest read-operation, some notion of a meaningful group of model elements is required. For that, MOIN offers the concept of model-partitions, which allows users splitting up the graphs represented by model content into manageable buckets.

The MOIN repository component basically deals with storing and loading of model-partitions.

[4] SAP and SAP NetWeaver are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Query Mechanism JMI is well suited for exploring models, by accessing attributes, following links etc. However, for many use-cases more powerful means of data retrieval are needed. The MOIN query API therefore provides flexible methods for retrieving model elements, based on their types, attribute values, relationships to other model elements etc.

As part of the query API, we expose the abstract syntax of a query language, which is specific for MOIN, the so called MOIN query language (MOIN QL). This query language is tailored to the needs of the MOIN clients and supports MOIN-specific concepts like model-partitions, which are not part of OCL. However, it is planned to implement a mapping mechanism, which is capable of translating query parts of OCL statements into MOIN QL.

Eventing Framework Events can be used by MOIN clients to receive notifications for e.g. changes on models. This supports an architecture of loosely coupled components. More specifically, events are raised upon a. creation or deletion of model elements, b. attribute changes on a model element (i.e. value change, add or remove attribute), c. creation or deletion of links or d. creation, storing or deletion of model partitions, e. membership change of model elements wrt model-partitions etc.

Commands The MOIN command framework is the basis for undo/redo functionality, which will be part of most of the modeling tools. For that, modeling tools perform operations on the model content, which is bundled into a command object. Command objects can be put onto a stack, the so called undo/redo-stack. All commands implement methods for undo and redo. By calling undo for commands on the undo/redo-stack, any given number of user-actions following a save-operation can be reverted.

Model Transformation Infrastructure (MTI) The model transformation infrastructure (MTI) is planned as basis for model-to-model and model-to-text transformations, which can be defined by MOIN users for specific meta-models. MTI will provide a framework for defining and executing these transformations, where OCL is considered as option for describing query parts of transformation rules. However, since the design of MTI is not final yet, a sound assessment on the suitability of OCL for these use cases in the context of MOIN can not be made.

MOIN Core The MOIN core is the central component in the MOIN architecture, implementing and enforcing MOF semantics. It is independent from the deployment options and development infrastructure aspects and calls the other components for implementing all of MOIN's functionality.

By managing the object instances, representing model elements, the MOIN core can also be seen as in-memory cache for model content. However, it also manages the complete lifecycle of objects, triggers events, is involved in the execution of commands and uses the repository layer to read or write data.

OCL Components Apart from query and probably MTI, OCL is primarily used in meta-models to state constraints for the associated models, which cannot be expressed with means of MOF directly.

For the efficient evaluation of constraints, mainly four components are essential: a. the OCL parser, b. the OCL evaluator, c. the impact analysis and d. the MOIN core, which manages the checking of constraints by calling the impact analysis and the OCL evaluator.

The impact analysis is essential for the efficient implementation of constraint checking, as it avoids the unnecessary evaluation of constraints in specific situations. The impact analysis is described in section 4 in more detail.

3 Related Work

To our knowledge, there is not much related work in the area of optimization of OCL expression evaluation at the moment. In [6] the authors describe an algorithm to reduce the set of OCL constraints that have to be reevaluated if a model change did occur. Our solutions is based on that approach, but is more general in that it covers any sort of OCL expression, whereas [6] focuses on constraints only, which makes further optimizations possible: Assuming that all constraints are initially valid (i.e. works start with a consistent model) it is sufficient to identify only the events potentially causing constraint violation. However, there are application scenarios in MOIN where this assumption does not hold at all. Imagine a tool, which checks constraints while a user is editing, and marks elements violating constraints. Here it is important to know whether the violation was fixed so the highlighting can be removed. Sometimes, it may even be desirable, or at least tolerable to temporarily leave meta-models in an inconsistent state, like situations where the architect or designer is not yet able to provide all mandatory information. In other scenarios, like model transformations, we have to deal with any sort of OCL expressions, not only constraints, thus any modification causing the query to evaluate differently are relevant.

In a second paper [7], the same authors describe a method to reduce the number of context instances for which relevant OCL constraints have to be evaluated. It can be seen as a further optimization on top of the approach in [6]. The idea of decomposing expressions into sub-expression and building paths through the model was taken from there. However, there are two major differences: Firstly, the introduced method relies on an extension of the constrained model for computing instances. Helper associations are introduced to keep track of the applied navigations and to finally compute the affected instances, which have to be considered for reevaluation. This approach clearly violates one of our requirements that meta-models should stay intact avoiding any modifications not intended by the user. As we will see in the next section, we have taken a different approach to compute the relevant instance set which we believe is more flexible and allows further optimizations. Secondly, [7] leaves out some important details about how to handle indirect sub-expressions, esp. in the context of loop expressions (like `collect` and `iterate`), user-defined operations and attributes. Aiming at a full

support of all language features, we have to provide solutions for those topics as well.

In [8], the authors go even one step further, and actively rewrite constraints for further optimizations. This may even lead to attaching a constraint to a new context. While this approach may definitely lead to a better performance than our approach, we did not consider optimizations in that direction, because this would introduce additional management overhead if we hid that transformation from the modeller and kept the two versions of constraints in sync.

In [9] a rule-based simplification of OCL constraints is introduced. These simplifications involve constant folding, removing of tautologies and unnecessary if-then-else constructs and more and are used to simplify automatically generated OCL constraints. We intentionally abandoned that approach in our work, because of usability issues: Users may get totally confused if, e.g. violations are reported on expressions they have never seen before, but merely result from silent rewrites of their expression by the underlying infrastructure.

4 OCL Impact Analysis in the SAP Modeling Infrastructure

This section presents the architecture and functionality of the OCL impact analysis and how it fits into SAP's modeling infrastructure.

4.1 Architecture

To support a wide range of different usage scenarios we decided to implement the *impact analyzer* (IA) as a general optimization add-on to applications[5], which have to deal with OCL in some way.

As indicated in Figure 1, interacting with the IA happens in two phases: Firstly, in the *analysis* phase (steps 1-3), a set of parsed OCL expressions is passed to the IA, whereupon a filter expression is returned. This filter can then be used to register with the eventing framework, so the application will only be notified about relevant model change events. Secondly, in the *filter* phase (steps 4-6), a received event can be forwarded to the IA to identify the OCL expressions affected by a change and the set of context instances per expression, for which the expression has to be reevaluated.

In fact, IA does not actually return a set of context instances, but OCL expressions evaluating to that set. This allows for quick responses and leaves further optimizations to the evaluator. Furthermore, in contrast to [7], this approach does not rely on an extension of the meta-model.

Typically, steps 1-3 are only executed once at the beginning, whereas steps 4-6 are executed often during the run-time of an application using IA.

[5] Such an application could be a constraint checker, a model transformation engine, etc.

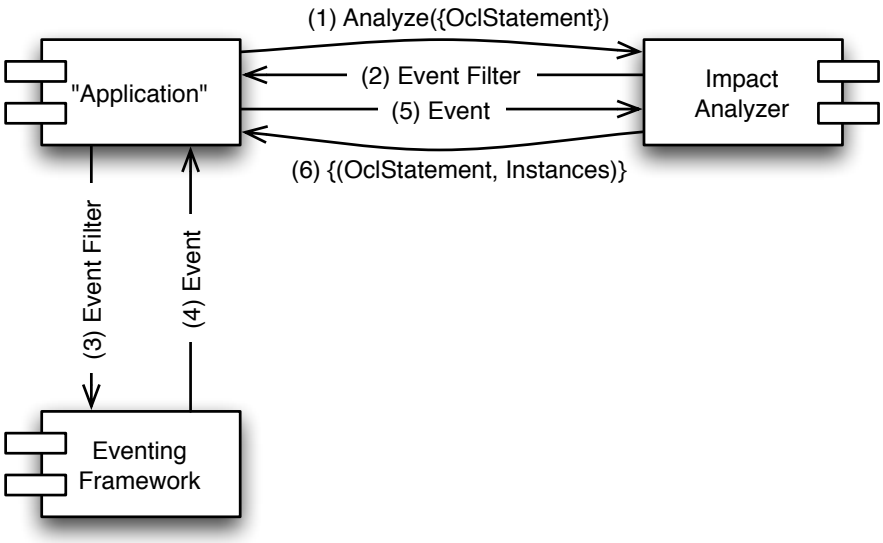


Fig. 1. Impact Analyzer Architecture.

During the analysis phase, the OCL expressions are analyzed and internal data structures are built up, which are then used in the filter phase for quick look-ups. These data structures are based on so-called *internal events* which represent classes of *model change events* provided by MOIN's eventing framework. The relationship between internal events and model change events is shown in Table 1.

The analysis phase itself is split up into a *class scope analysis* and a subsequent (optional) *instance scope analysis*. Both methods are described in the following sections.

4.2 Class Scope Analysis

The goal of the class scope analysis is to find the set of internal events (i.e., all types of model change events) which can cause the given expression to evaluate differently. At this point, it is assumed that all affected expressions have to be evaluated for all its context instances[6]. As outlined in Section 3, we use a generalized approach from [6] and walk the abstract syntax tree (AST) representing the given OCL expression in a depth-first manner, tagging each node with internal events that are relevant to it:

– Variable expressions referring to `self` → `CreateInstance(context)`, where context identifies the type of `self`

[6] Hence the name *class scope* analysis.

Internal Event	Model Change Event
CreateInstance(c:MofClass)	ElementAddedEvent(o:RefObject) where `o.refMetaObject = c`
DeleteInstance(c:MofClass)	ElementRemovedEvent(o:RefObject) where `o.refMetaObject = c`
AddLink(e:AssociationEnd)	LinkAddedEvent(f:RefFeatured, link:Sequence(RefObject)) where `e = f`
RemoveLink(e:AssociationEnd)	LinkRemovedEvent(f:RefFeatured, link:Sequence(RefObject)) where `e = f`
UpdateAttribute(a1:Attribute)	AttributeValueEvent(RefObject o, Attribute a2) where `a1 = a2`

Table 1. Mapping between internal events and model change events

- Operation calls `C.allInstances()` → `CreateInstance(C)`, `DeleteInstance(C)`
- Association end calls to `aE` → `AddLink(a)`, `RemoveLink(a)`, where `a` identifies the association to which the association end `aE` belongs
- Attribute call expressions to `a` → `UpdateAttribute(a)`

An OCL expression needs to be reevaluated if an event occurs which matches one of the internal events attached to a node in its AST. The IA's internal data structure therefore associates each internal event with a set OCL expressions affected by a model change event matching that internal event. Given a concrete model change event during the filter phase, IA determines the corresponding internal event and simply looks up the OCL expressions affected by that event.

For user-defined attributes and operations, the analyzer recurses into their bodies. The evaluation of a user-defined attribute or operation changes if its body is affected by a change to the model, thus affecting the evaluation of any expression referring to that user-defined operation or attribute.

```
inv: context Department
self.employee->select(e|e.age<23)->size()<self.maxJuniors
```

Listing 1.1. Example OCL expression used throughout this paper.

Example: Given the OCL expression in Listing 1.1, stating that per department only a certain number of junior employees are allowed. After having applied class scope analysis, the following internal events are relevant to the given expression: `CreateInstance(Department)`, `AddLink(employee)`, `RemoveLink(employee)`, `UpdateAttribute(age)`, `UpdateAttribute(maxJuniors)`.

4.3 Instance Scope Analysis

The goal of instance scope analysis is to reduce the number of context instances for which an expression needs to be reevaluated. Following the approach in [7], this is done by identifying navigation paths (i.e., sequences of attributes and association ends) in an expression starting at the context. If an object is changed, an OCL expression has to be reevaluated for those context instances from where the changed object can be reached by navigating along these paths. Given an element affected by a change, the set of context instances for which an expression needs to be reevaluated, can be found by following the reverse of the navigation paths. Once identified, these reverse paths are turned into OCL expressions and stored in the internal data structure. By evaluating these expressions, the set of context instances can be computed from a given changed element.

The following sections describe in more detail how sub-expressions and subsequently navigation paths can be identified and how they are reversed and translated into OCL.

Identifying Sub-Expressions The first step is to find sub-expressions. Sub-expressions start with a variable, or `allInstances()` and end in a node being the source of an operation with a primitive return type or in a node being a parameter of an operation or the body of a loop expression. Sub-expressions can also contain child sub-expressions in the body of a loop expression.

Two types of sub-expressions can be distinguished: *class* and *instance*.

Class sub-expressions start (directly or indirectly) with `allInstances()` and thus have to be evaluated for all instances of a class. There is no way to reduce the number of instances in this case.

Instance sub-expressions on the other hand start (directly or indirectly) with `self`. In this case, a subset of context instances can be identified for which the expression has to be reevaluated. The following steps only apply to instance sub-expression.

Example: Given the OCL expression in Listing 1.1, the following sub-expressions can be identified:

1. `self.employee->select()`
2. `e.age`
3. `self.maxJuniors`

Identifying Navigation Paths As per definition, sub-expressions consist only of navigation operations but do not necessarily start at the context. To get a sequence of navigation operations starting at the context, the navigation contained in a child sub-expression has to be concatenated with the navigation of the parent sub-expression. This approach only works for loop expressions calculating a subset of their source (e.g. select, reject).

```
inv: context Employee
self.employer->collect(d:Department|d.employee)->...
```

Listing 1.2. The body of `collect` has to be included in the parent's navigation path.

Example: Considering the last example, the corresponding navigation paths, relative to the context, are:

1. <employee>
2. <employee, age>
3. <maxJuniors>

As the second sub-expression does not start at the context, its navigation path has to be concatenated with the navigation path of its parent, i.e., the first sub-expression.

For loop expressions returning a completely different set compared to their source (e.g. collect, iterate), another approach has to be used. In this case the navigation contained in the body has to be included in the parent's navigation path because it contains vital information about how to get from the source type to the return type. Otherwise, there would be a gap in the navigation path.

Example: Considering the OCL expression in Listing 1.2, the following two navigation paths can be identified:

1. < employer, employee, ... > (for the parent sub-expression)
2. < employer, employee > (for the child sub-expression)

In this case, the `collect` operation takes a set of Departments and returns a set of Employees. Only by examining the body it can be said how to get from Department to Employee: by following the `employee` association end.

Reversing Navigation Paths Having applied class scope analysis before, each AST node has internal events, by which it is affected, attached to it. For each AST node and each internal event attached to it, the way back to the context has to be identified. This is done by reversing the navigation path which leads from the context to the currently considered AST node.

Example: Continuing the running example in Listing 1.1, we get the reverse navigation paths for each relevant internal event identified by class scope analysis as shown in Table 2.

If a new Department is created, the expression obviously has to be evaluated for that Department, therefore, the reverse navigation path is empty. If an employee is added to or removed from a department, the reverse navigation path is empty as well. More interesting is the case when the age of an employee is changed. In this case, navigating along the *employer* association end (opposite of employee) reveals the department, for which the expression has to be reevaluated.

Internal Event	Navigation Path
CreateInstance(Department)	<>
AddLink(employee), RemoveLink(employee)	<>
UpdateAttribute(age)	< employer >
UpdateAttribute(maxJuniors)	<>

Table 2. Internal events and corresponding navigation paths.

Translating into OCL The identified reverse navigation paths are translated into OCL and then stored in the internal data structure. Each internal event is associated with a number of expressions, which are affected by it. For each pair of internal event and affected expression a set of OCL reverse path expressions is associated with it. Evaluating that set of expressions for a given changed object results in the set of affected context instances.

For navigating along association ends, translation is straight forward: An association call expression is created referring to the reversed, i.e., opposite association end. Reversing object-valued attributes, however, is not that easy. Unfortunately, OCL does not offer a construct to find the owner of an attribute value. However, a legal OCL expression can be constructed which finds the attribute value's owner. The construct simply goes through all instances of a type `T` and checks whether it's attribute `attr` points to the given value `v`.

```
T.allInstances()->select(attr=v)
```

For performance reasons, an optimized evaluator could simply replace such a construct by a `v.immediateComposite()` call on the JMI object to determine the value's owner.

Example: Continuing the running example, in case of an `UpdateAttribute(age)` event the following OCL expression computes the Department for which the expression in Listing 1.1 needs to be reevaluated.

```
inv: context Employee
    self.employer
```

5 Preliminary results

To show the efficiency of our approach we present a theoretical best and worst case analysis, assessing the whole range of possible performance benefits. Furthermore, to see the impact on performance in practice we set up a test scenario using the MOF constraints defined in [2] with the UML meta-model as an instance of MOF.

5.1 Theoretical analysis

How much can be gained from using IA depends on four dimensions: The meta-model, the set of constraints, the set of instances and the set of events reported

to an application. Note that these dimensions are not entirely orthogonal to each other.

Worst case analysis The first optimization used by IA is to reduce the number of events by removing irrelevant events. If we assume that only relevant events are reported, then there is no benefit from using IA. The next step is to reduce the number of expressions to be considered for evaluation to a subset of relevant expressions. If we further assume that all, or at least a large number of constraints use the same features, then each event is relevant to all expressions. So, there is no benefit either. The last optimization step it to reduce the number of context instances for evaluation. Assuming all expressions are class expressions (i.e., use `allInstances()`), this last optimization step does not yield performance benefits either. Hence, the worst case is that there are no performance benefits at all. Even worse, analyzing the set of expressions and computing the affected context instances adds an extra penalty to the application using IA.

Best case analysis In the very best case we assume that only irrelevant events occur. Without any optimizations lots of unnecessary work has to be done, whereas an application using IA is not even bothered – no unrelated events are reported to the application due to the filter mechanism provided by MOIN.

The second best case is that relevant events occur. If we assume that each feature in the model is only referred to by one statement, each event is only relevant to one statement. Given n statements, only $1/n$ of the statements have to be considered for evaluation. Furthermore, if we assume, that the model makes only use of one-to-one associations, for each model element affected by a change, only one context instance has to be considered for evaluation. In total, given n OCL statements and m instances per context, the work can be reduced to $\frac{1}{nm}$ of the work of an application without IA support. Depending on the number of statements and instances, this can be an enormous reduction.

5.2 UML-meta-model + MOF-constraints

To have a more realistic assessment of the performance benefits achieved by IA, we used the MOF-constraints[7] and the UML-meta-model, an instance of MOF, as a test scenario. Both, the MOF-constraints and the UML-meta-model are taken from real life and are non-trivial. As a reference, we shall compare all results to a *naive application*, i.e. an application which has no means of optimization and has to reevaluate all constraints for any change to the model.

There are two different applications using IA to different extents. Firstly, *class scope application*, which uses only the class scope analysis part. Secondly, *instance scope application*, which uses IA to its fullest extent. All three applications are exposed to the same model changes in the same order. Each model

[7] As not all of the constraints where proper OCL and our evaluator did not support type conversion at the time, we used only a subset of 38 constraints.

element in MOF is covered by exactly one event if there exists an instance of that model element in the UML-meta-model.

Reduction of expressions We consider the number of expressions which have to be evaluated after an event has been reported. In Figure 2 we compare the results from the *class scope application* to the *naive application*[8]. As the naive application has no means of optimization, it has to evaluate all (38) expressions for any event, whereas the class scope application does not have to evaluate expressions which cannot have changed due to the reported event.

For about 1/4 of the events, the number of relevant expressions could be reduced to one by applying class scope analysis. This is a reduction by 97%. For about 1/8 of the events, the number of relevant expressions could only be reduced to 12 and 11 respectively. Still, this is a reduction by 68% (71%). In average, the number of expressions to evaluate was reduced by 88%. The Median is 92%. As instance scope analysis does not reduce the number of expressions further, it is omitted.

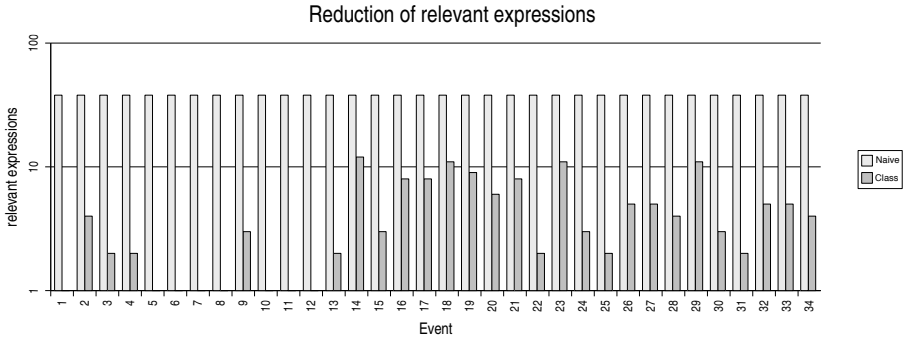


Fig. 2. Reduction of relevant expressions.

Reduction of context instances Here we consider the number of evaluator calls necessary to evaluate all affected expressions. An evaluator call is equal to the evaluation of one expression for one context instance. The numbers in Figure 3 also include evaluator calls necessary to compute the set of affected context instances. As class scope analysis reduces the number of expressions to evaluate, the number of evaluator calls is reduced as well. Therefore, the number of evaluator calls experiences about the same reduction as the number of expressions. After an already substantial reduction by class scope analysis, instance

[8] As class scope analysis does not reduce the number of expressions to be considered for reevaluation, it is not included in the chart. The results are equal to *class scope application*.

scope analysis achieves another enormous reduction: From several thousands to twenty or less for about 77% of the events (compare Figure 3). In total, the number of evaluator calls was reduced by three to four orders of magnitude, which is an enormous benefit in performance. In contrast to that, the naive application has to do some 26 000 evaluator calls per event to check the consistency of all constraints.

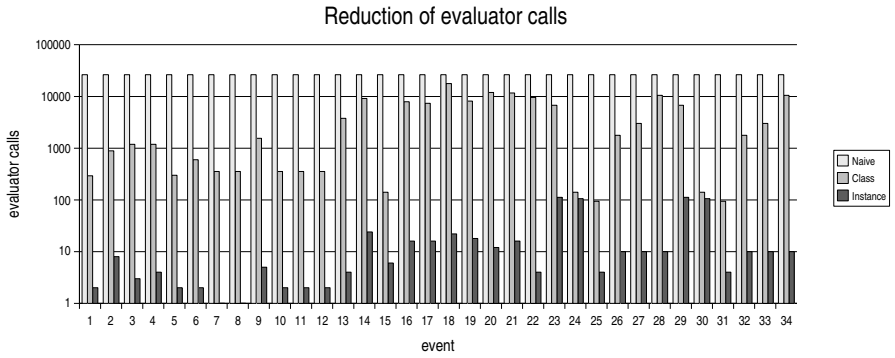


Fig. 3. Reduction of evaluator calls, including evaluator calls for computing the set of relevant context instances.

6 Conclusion

While efficient support for OCL is considered crucial in large-scale modeling environments, surprisingly little work has been published on optimizing OCL expression evaluation in case of arbitrary model changes. In this paper, we have reported on our experiences with integrating OCL into SAP's next generation modeling infrastructure MOIN.

Although some of the basic approaches from literature could be reused [6, 7], the actual implementation had to divert from these methods to cope with the (non-)functional requirements pertinent to MOIN. Most notably, we currently refused to implement any techniques that would result in silent or user-invisible changes to either the meta-models or the OCL expressions related to those meta-models. We know that this may lead to sub-optimal results in terms of performance, but preliminary experimental results show that the implemented techniques can still lead to a significant and hopefully sufficient performance gain. Further optimization techniques may be considered in the future, but they will have to be evaluated carefully on their trade-offs regarding other desired features.

Another path of optimization that we have not fully explored yet is the way how context instances are computed. The current approach is built on using

OCL as the expression language, but we plan to investigate using the internal MOIN Query Language for speeding up this computational step.

7 Acknowledgements

We would like to thank our colleagues Kristian Domagala, Harald Fuchs, Hans Hofmann, Simon Helsen, Diego Rapela, Murray Spork, and Axel Uhl for fruitful discussions during the design and the implementation of the OCL Impact Analyzer.

References

1. Object Management Group: MDA Guide. June 2003.
2. Object Management Group: Meta Object Facility (MOF) Specification. April 2002. http://www.omg.org/docs/formal/02-04-03.pdf.
3. Object Management Group: OCL 2.0 Specification (ptc/2005-06-06). June 2005.
4. Object Management Group: UML 2.0 Superstructure Specification (pct/03-08-02). August 2003.
5. Object Management Group: MOF 2.0 Query / Views / Transformations - Request for Proposal. http://www.omg.org/docs/ad/02-04-10.pdf, 2002.
6. Cabot, J., Teniente, E.: Determining the Structural Events that May Violate an Integrity Constraint. In: Proc. 7th Int. Conf. on the Unified Modeling Language (UML'04), LNCS, 3273 (2004) 173-187
7. Cabot, J., Teniente, E.: Computing the Relevant Instances that May Violate an OCL constraint. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'05), LNCS, 3520 (2005) 48-62
8. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'06), June 2006.
9. Giese M., Hähnle R., Larsson, D.: Rule-based simplification of OCL constraints. In: Workshop on OCL and Model Driven Engineering at UML2004, pages 84-89, 2004.

Integrating OCL and Model Transformations in Fujaba

Mirko Stölzel[1], Steffen Zschaler[1], and Leif Geiger[2]

[1]Dresden University of Technology, Department of Computer Science
steffen.zschaler@inf.tu-dresden.de
http://st.inf.tu-dresden.de/

[2]Universität Kassel, Wilhelmshöher Allee 73, 34121 Kassel
leif.geiger@uni-kassel.de
http://www.se.eecs.uni-kassel.de/se/

Abstract. This paper discusses the integration of the Dresden OCL Toolkit into the Fujaba Tool Suite. The integration not only adds OCL support for class diagrams but also makes OCL usable in Fujaba's model transformations. This makes Fujaba's model transformations more powerful, completely platform independent and easier to read for developers who are already familiar with OCL. By using the code generator of the Dresden Toolkit, we are able to generate executable Java code from Fujaba's model transformations including the OCL constraints.

1 Introduction

The Fujaba Tool Suite [1] is a CASE tool which supports Model Driven Development (MDD) [2]. Within MDD model transformations play an important role. Fujaba offers special interaction diagrams to specify model transformations. Within these diagrams most of the transformations are specified graphically. Nevertheless, some expressions have to be specified textually, like complicated constraints, return values, etc. Since Fujaba generates Java source code from model transformations, these textual statements have been specified using Java expression. Currently, no syntax-checking is done for these expressions, so an erroneous expression results in a compile error after code generation. Newer work adds C++ code generation to Fujaba. Note, that if a developer wants to use C++ code generation, the constraints have to be written in C++ syntax.

So, it would be helpful to have a platform independent constraint language, which makes syntax checking possible within Fujaba's model transformations, adds code completion and code generation for the different target languages Fujaba offers. This work suggest to use the Object Constraint Language (OCL) [3] for this task. We have integrated the Dresden OCL Toolkit [4] into the Fujaba Tool Suite. So, we use OCL as constraint language for Fujaba's model transformations.

Section 2 briefly describes how model transformations are specified using Fujaba, Section 3 describes the integration of OCL into Fujaba's model transformations, Section 4 discusses code generation and Section 6 concludes.

2 Story Diagrams – A Short Overview

The Fujaba Tool Suite [1] uses Unified Modelling Language (UML) [5] class diagrams to model the structure of an application. A previous work [6] has already integrated the Dresden OCL Toolkit [4] for use in Fujaba's class diagrams. For behavior specification, model transformation are specified by using graph transformations within Fujaba. This is done by modelling specialized UML interaction diagrams for the method bodies, so called story diagrams [7, 8]. From such diagrams Fujaba can then generate executable Java source code.

Figure 1 show such a story diagram. The activity diagram models the control flow. The graph transformations within the activities model the behavior. The first activity of Figure 1 shows such a graph transformation. Here, starting from the object `this`, which is the object the method `nameExists()` is called on, a child is search via the `children` association. This child's `name` attribute should equal the passed `name` parameter. If such a child is found, it is stored in a local variable called `child`.

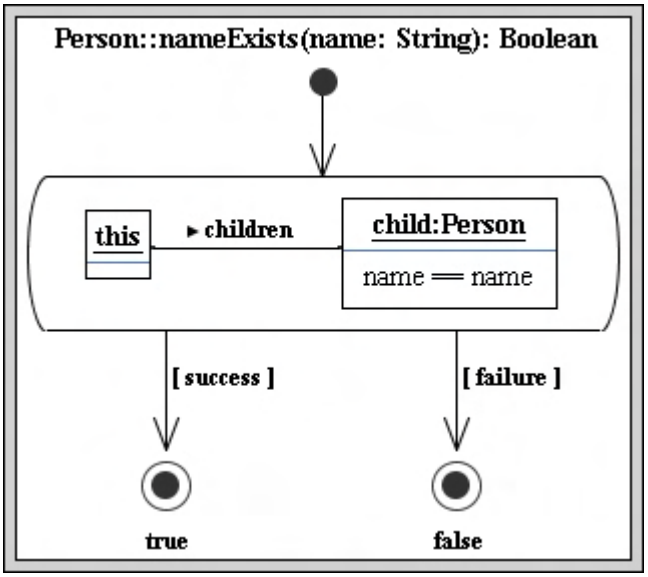


Fig. 1. Story diagram

Afterwards, the activity is left. If the graph transformation was applied, it is left via the `success` transition. So the method returns `true`. Otherwise, the `failure` transition is taken, thus `false` is returned.

Note, that in Fujaba's story diagrams, there are several places, where Java source code can be used, e.g. the return value for a stop activity can be any Java expression and is directly copied into the source code during code generation.

3 Integrating OCL into Story Diagrams

This section discusses the integration of OCL into Fujaba's story diagrams. At first it will give you a short overview of the possibilities to use OCL in story diagrams. Then some special characteristics of Fujaba's story diagrams, which must be considered to use OCL in story diagrams, are presented. Finally a possible solution which considers these special characteristics will be shown.

3.1 Where to Integrate

In this section, we will present a short overview of all possibilities to use OCL in Fujaba's story diagrams. On the left side of the Figures 2–6 one can see some examples with the actual notation of Fujaba while on the right the same example is illustrated using OCL:

Attribute expressions can be used to assign new attribute values to an attribute of an object and to define some additional attribute conditions which must be fulfilled by an object. In the example of Figure 2 the value of the `name` attribute of the `this`-object is assigned to the `name` attribute of the `child`-object by calling the `getName()` method of the `this`-object. On the right side of Figure 2 one can see that the `name` attribute of the `this`-object can be directly referenced using OCL.

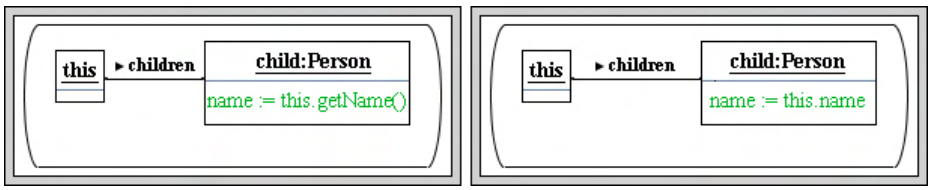


Fig. 2. Attribute assertion and attribute constraints

Collaboration statements are used to execute methods, to define new variables or to assign new values to variables. These operations can be combined using the sequential, if- or while composition. In the following example (Figure 3) a collaboration statement is used to define the variable `count` of type Integer. The `sizeOfChildren()` method is an automatic generated method of Fujaba for the to-n association children. It returns the number of `Person` instances which are assigned to the `this`-object as a child. Using OCL one can reference the children association directly and can call the `size()` method of OCL-Set to get the number of children of the `this`-object.

Additional constraints are boolean constraints which can be assigned to a story pattern so that the story pattern is applicable if the constraint evaluates to true. In the example of Figure 4 the additional constraint defines that the `this`-object must have exactly five children.

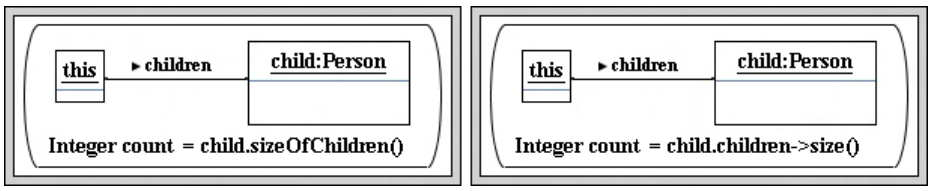


Fig. 3. Collaboration statements

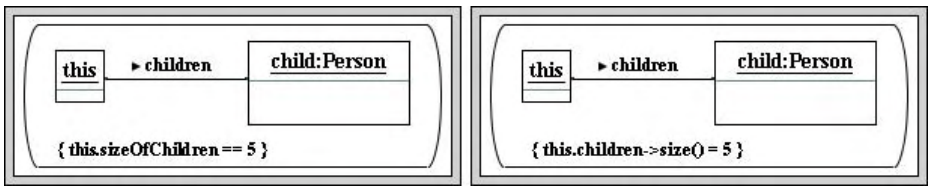


Fig. 4. Additional constraints

Boolean transition guards can be used to realize a if- or while-composition in the activity diagram part of the story diagrams. In the following example the variable `found` will be set to true if the `child`-object was successfully bound in the previous story pattern. As one can see on the right side of this example the `oclIsUndefined()` method can be used to formulate the boolean condition with OCL.

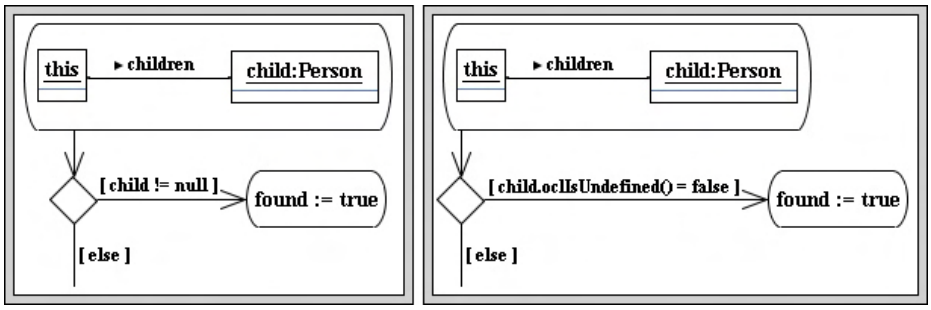


Fig. 5. Boolean if-condition

Method return value The last possible use of OCL in Fujaba's story diagrams is represented in Figure 6. There one can see that you have the possibility to provide a return clause for a ***stop activity*** of a story diagram.

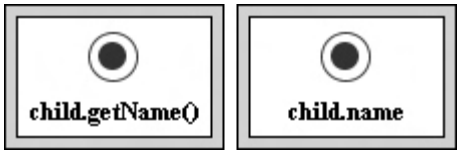


Fig. 6. Stop activity

3.2 Resolving Scoping

To integrate OCL in Fujaba's story diagrams we use the OCL parser of the Dresden OCL Toolkit. It checks the syntax and the consistency of an OCL constraints in the context of the containing story diagram. To perform a consistency check the parser of the Dresden OCL Toolkit tries to find all variables which are referenced within an OCL constraint in its story diagram. To do so, the parser has to know which variables and objects are defined in the corresponding story diagram. So we have to generate the context of the OCL constraints in a story diagram.

When generating the context of OCL constraints in Fujaba's story diagrams we have to consider some special characteristics of story diagrams:

- In story diagrams the `this`-object and the method parameters are predefined bound objects. Those can always be referenced in OCL constraints.
- A story diagram in general contains many execution paths. Every path visits differnt story activities and so different variables and objects can be bound. It can e.g. occur that one variable is not initialized on one special path leading to a story activity and initialized on another one. That's the reason why only these variables and objects can be used in an OCL constraint of a story activity which are defined on every paths leading to that story activity.
- An object of a story diagram is initialized with a valid value if the corresponding story pattern is applicable. So the objects of an story pattern can only be referenced by the OCL constraints of the next story activity if the story activity is connected by a success or eachtime transition. An eachtime transition is used in combination with a so called foreach activity. This special activity is not left after the first object was found, but the specified transformations are executed on every valid object allocation. In the example of Figure 1 we could have use a foreach activity to count all children where the `name` attribute equals the passed `name` parameter.

In the following we present an algorithm which considers these characteristics and can be used to generate the context of an OCL constraint in a story diagram. The context, called environment, of an OCL constraint contains all variables that are visible for the OCL expression. To obtain this context, an environment is assigned to every element in the story diagram, beginning with the start activity. An environment encapsulates a set of name–type bindings representing the variables accessible under this environment. When a name lookup occurs, the environment first checks whether it contains a corresponding binding itself. If this is not the case, the environment can delegate the lookup to its parent environments (other environments linked to it via a parent association). If

all parent environments agree on the result of the lookup, this will be returned. If they do not agree, the lookup fails. As we will see, parent–child relations can, thus, be used to represent the control flow in a story diagram. Note that story diagrams allow to the deletion of objects from the object graph. Therefore, after deletion of an object its name will be no longer bound. To represent this, environments distinguish different types of bindings; one of them is used to mark deleted objects.

In order to clarify the context generation algorithm an example story diagram is represented in the next figure. There one can see that first an initial environment e1

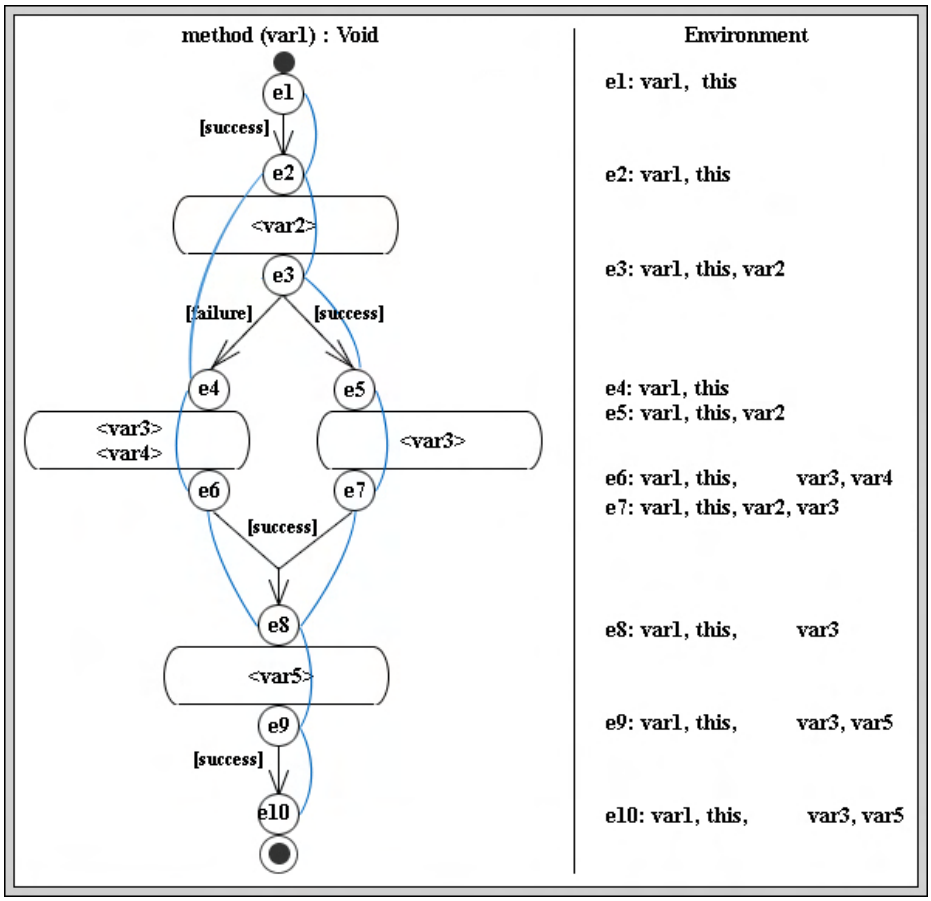


Fig. 7. Generation of the OCL-Context

is assigned to the start activity of the story diagram and that the this-object and the method parameter var1 are added to this environment. In the next step, the outgoing transition of the start activity is traversed and the first activity is visited. In addition, the

environment e2 is assigned to the activity as input environment. Since the variables this and var1 of the environment e1 can also be used within the first activity, the parent/child relationship between e1 and e2 is created. In the first activity the variable var2 is created and it is added to the outgoing environment e3 of the activity.

In the next step the two outgoing transitions of the second activity are traversed and the environments e4 and e5 are assigned to the corresponding story activities. It must be considered, that on the path following the failure transition the story pattern of the second activity was not applicable. Consequently, we cannot assume that the objects of the second activity were successfully bound. Therefore these objects cannot be used in following OCL constraints. That's the reason why the parent/child relationship is made between the environment e4 and the environment e2 and not to the environment e3. Similarly, the variable var2 could successfully be bound when taking the success transition and can be used in following OCL constraints. So the parent/child relationship between the environment e3 and e5 is created. In the next steps the environment e6 and e7 are created, which contain the visible variables, and the outgoing transitions are traversed.

As result the environment e8 is assigned to the next story activity and the parent/child relations between the environment e8 and the environments e6 and e7 are created. At this point the second problem mentioned above must be considered. Since the variable var4 is defined only on the left path, the environment e8 does not contain this variable. The same problem applies to the variable var2 also. Because of the success transition this variable can be used only in the right path and thus the variable var2 is also not a part of the environment e8.

The last step of the generation process is to generate the environments e9 and e10 which is assigned to the stop activity of the story diagram.

3.3 Fujaba and the Dresden OCL Toolkit

Fujaba4Eclipse[9] is a Eclipse Plugin that among other functions integrates Fujaba's story diagrams into eclipse to specify methods. On the basis of Fujaba4Eclipse the integration of the Dresden OCL Toolkit[4] for Fujaba's class diagrams has already been accomplished in [6].

We are, presently, extending this integration to also cover story diagrams. Thus, input, consistency and syntax checking of OCL constraints in story diagram should be possible, as it is possible already for Fujaba's class diagrams. We use the algorithm described in Section 3.2 to generate the context of OCL constraints in a story diagram. The generated context is used by the parser of the Dresden OCL Toolkit to check whether referenced variables within an OCL constraint are defined in the corresponding story diagram.

Figure 8 shows a screen shot of the tool. In the left lower part of this figure you can see the OCL-Editor for Eclipse which allows you to create and edit OCL constraints for a given story diagram. Additional you can use the OCL parser of the Dresden OCL Toolkit to check syntax and consistency of the OCL constraints against the story diagram. In the example shown in Figure 8 one can see, that an error message is shown in the problems view of eclipse, since the variable var4 is not defined on the right path of the example story diagram.

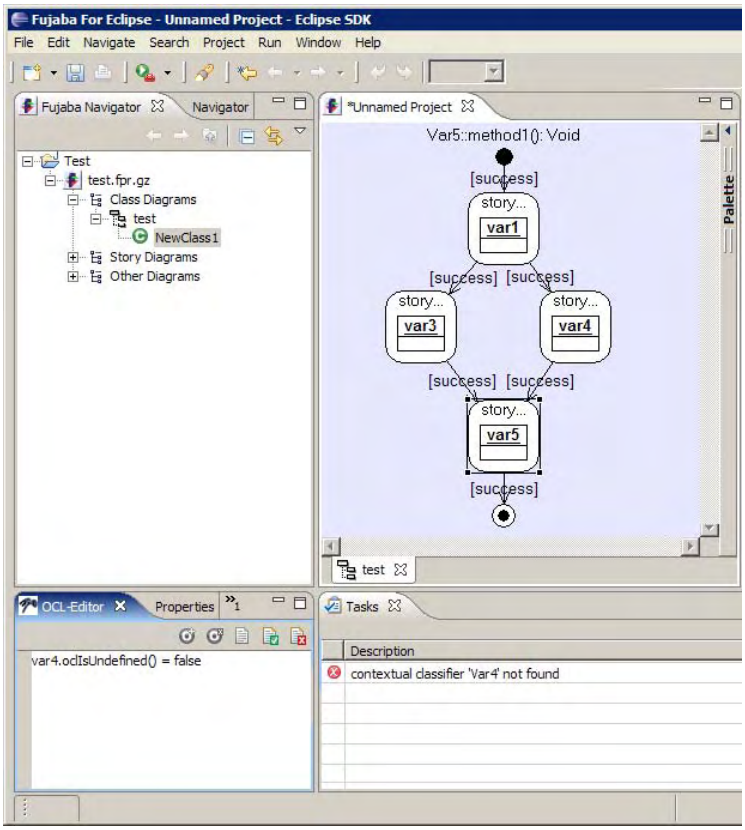


Fig. 8. Generation of the OCL-Context

4 Generating Code

As already mentioned, Fujaba generates executable Java code from class diagrams and model transformations. The code that would be generated for the left hand side of Figure 4 is shown below.

```
01 // bind child: Person
02 Iterator iter = this.iteratorOfChildren ();
03 while ( !(fujaba__Success) && iter.hasNext () )
04 {
05    try
06    {
07       child = (Person) iter.next ();
08       // check isomorphic binding
09       JavaSDM.ensure ( !(this.equals (child)) );
10       // constraint
```

```
11       JavaSDM.ensure ( child.sizeOfChildren() == 5 );
12       fujaba__Success = true;
13    }
14    catch ( JavaSDMException e ) {}
15 }
```

To search through all children of the `this` object, a `Iterator` is created in line 02. The while loop from line 04 to line 15 is repeated till one child has been found, that matches all conditions (`fujaba__Success == true`) or till no more child exists in the list. In this loop, in line 07 the current `child` object is fetched from the list. Since the `this` object, and the `child` object are both of class `Person`, it is possible to make a person its own child. Our semantics forbids that (if not stated different), so this is checked in line 09. Note, that Fujaba provides the library method `JavaSDM.ensure(boolean)` which simply does nothing, when passed true and throws a `JavaSDMException` otherwise. So, if `this` equals `child`, this would end the checks for the current object and continue with the next one. Otherwise the additional constraint is checked in line 11. Note, that the text from the constraint is directly copied into the code surrounded by another `JavaSDM.ensure`. If this test is also passed, `fujaba__Success` is set to true, to indicate that a valid child has been found. The loop is terminated in that case.

If the additional constraint is now specified in OCL, as done in the right hand side of Figure 4, the code generation has to be adapted. We are currently integrating the code generation of the Dresden OCL Toolkit in the presented work. The modified code generation would leave most of the code above untouched, but changes only the check of the condition in line 11. The source code below shows the code which is generated.

```
01 //bind child: Person
02 Iterator iter = this.iteratorOfChildren ();
03 while ( !(fujaba__Success) && iter.hasNext () )
04 {
05    try
06    {
07       child = (Person) iter.next ();
08       // check isomorphic binding
09       JavaSDM.ensure ( !(this.equals (child)) );
10       //******************constraint***************
11       OclAny self =
12            (OclAny) Ocl.getOclRepresentationFor(this);
13       OclBoolean constraintValid=
14           self.getFeatureAsCollection("children").
15              size().isEqualTo( new OclInteger(5) );
16       JavaSDM.ensure ( constraintValid.isTrue() );
17       //******************constraint***************
18       fujaba__Success = true;
19    }
20    catch ( JavaSDMException e ) {}
```

```
21 }
```

Within the Dresden OCL Toolkit the OCL Standard Library is implemented by some Java classes, which are used by the Java code, created by the Java code generator of the Dresden OCL Toolkit, to evaluate an OCL constraint. To evaluate the OCL constraint of the right hand side of Figure 4 an instance of the class `OCLAny` is created as one can see in line 11 of the code example shown above. This instance is used in line 13 to get an instance of the class `OCLCollection` which represents the children association end of the `this`-object. Afterwards the number of the elements in this collection is determined using the `size()` method of the `OCLCollection` instance. This results in an instance of the class `OCLInteger` which `isEqualTo()` method is used to evaluate whether the number of the collection elements equals 5. As result of the `isEqualTo()` method call an instance of the class `OCLBoolean` is created which `isTrue()` method returns the result of the comparison. So the result of this method can be used as input of the `JavaSDM.ensure()` method call as one can see in line 15.

5 Related work

Many CASE tools offer OCL support for class diagrams. The Dresden OCL Toolkit e.g. was also integrated in Together and ArgoUML. But those tools have no support for model transformation and since no integration of OCL in other diagrams. The EMFT project [10] supports OCL for constraints and queries. One can use OCL for constraints on the static model and for specification of querying behavior. This way e.g. derived attributes can be modeled. So EMFT uses OCL for some very basic behavior specification. But it has no support for model transformations.

The QVT standard [11] by the OMG has some similar ideas. QVT defines a model transformation language which uses OCL. QVT extends the OCL with imperative expressions to make it more powerful. In this ImperativeOCL things like attribute assignments, link creation etc. can now be expressed. In our approach this imperative part is modeled using story diagrams. Since now, complete tool support for QVT is still missing.

6 Conclusions

The Fujaba Tool Suite is a CASE-Tool which supports the most important diagrams of the Unified Modelling Language with code generation for Java. To also specify the behavior of a system modelled with Fujaba one can use so called story diagrams.

As described in Section 2 story diagrams combine UML activity diagrams and collaboration diagrams for the specification of methods. Within story diagrams some expressions, like additional constraints, return values, etc are specified textually using Java expressions. These expressions are inserted identically in the code generated by Fujaba. If a developer wants to use another programming language than Java every constraint within the story diagrams have to be changed separately. So it is useful to specify the additional constraints using the Object Constraint Language.

Therefore, we discussed the possibilities to use OCL in Fujaba's story diagrams in Section 3 and described some special characteristics which must be considered to generate the context of OCL contraints within story diagrams. After that we explained an algorithm to generate the OCL context considering the special characteristics.

In Section 4 we described the code generation for Fujaba's story diagrams and discussed how the generated code of a story diagram could look like using OCL.

As already mentioned in Section 3, we use the Dresden OCL Toolkit to integrate OCL in Fujaba's story diagrams. This enables using OCL in various places in Fujaba's story diagrams, while maintaining the ability to generate code. Development of a prototype implementation of the concepts discussed in this paper is nearing completion.

References

1. Zündorf, A.: The fujaba toolsuite. http://www.fujaba.de/ (1999)
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model-Driven Architecture–Practice and Promise. Addison-Wesley (2003)
3. Object Management Group: UML 2.0 OCL specification. OMG document ptc/2003-10-14 (2003)
4. OCL Toolkit Team: Dresden OCL Toolkit homepage. http://dresden-ocl.sourceforge.net/ (1999)
5. Object Management Group: UML resource page. http://www.omg.org/uml/ (2003)
6. Stölzel, M.: OCL für Fujaba. Großer Beleg, Technische Universität Dresden (2005) In German.
7. Fischer, T., Niere, J., Torunski, L.: Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und story-driven modeling. Diplomarbeit, University of Paderborn (1998)
8. Zündorf, A.: Rigorous object oriented software development, habilitation thesis (2001)
9. Zündorf, A.: Fujaba for Eclipse. http://wwwcs.uni-paderbord.de/cs/fujaba/projects/eclipse/ (2001)
10. The Eclipse Foundation: Emft - eclipse modeling framework technologies. http://www.eclipse.org/emft/projects/ocl/ (2006)
11. Object Management Group: Mof qvt final adopted specification. http://www.omg.org/docs/ptc/05-11-01.pdf (2006)

Restrictions for OCL constraint optimization algorithms

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf

Budapest University of Technology and Economics
Goldmann György tér 3., 1111 Budapest, Hungary

Abstract. Efficient constraint handling is essential in UML, in metamodeling, and also in model transformation. OCL is a popular, textual formal language that is used in most of the modeling frameworks to express constraints. Our research focuses on the optimization of OCL handling. Previous works have presented algorithms that can accelerate the constraint validation by rewriting and decomposing the constraints and caching the model queries. Although these algorithms can be used in general, there are special cases, where additional restrictions apply. The aim of this paper is to present these refined restrictions and the extended optimization algorithms.

1 Introduction

Metamodeling techniques can describe the rules of Domain Specific Modeling Languages (DSMLs), but these descriptions mainly consist of topological rules only. The available model items, their attributes and the possible relations between the items can be defined, but these definitions have a tendency to be incomplete, or imprecise. For example, there is a resource editor domain for mobile phones. Here, it is useful to define the valid range for slider controls that cannot be accomplished using metamodeling techniques. Another example is a metamodel that defines a DSML with computer networks. A single computer can have input and output connections, but these connections use the same cable with maximum n channels. Thus, the number of maximum available output connections equals the total number of channels minus the current number of input channels. Such constraints cannot be expressed by metamodel rules.

The real need for constraints applies also to graph rewriting-based model transformation [1]. Here the Left Hand-Side (LHS) of the rewriting rules define the pattern to find in the host graph. Beyond the topology of the visual models, additional constraints must be specified. Model transformations constraining the pattern matching are very popular, they are used for example in QVT [2]. Additionally, dealing with constraints means a solution to several unsolved model transformation issue [1].

One of the most wide-spread approaches to constraint handling is the Object Constraint Language (OCL) [3]. OCL is a flexible formal language. It was originally created to extend the capabilities of UML [4], but due to its flexibility, it can also be used in metamodeling environments with minor extensions [5].

Nowadays OCL is becoming essential both in metamodel-based model validation and model transformations.

Visual Modeling and Transformation Systems (VMTS) [6] is an n-layer metamodeling and model transformation tool. VMTS uses OCL constraints in model validation and also in the graph rewriting-based model transformation [1]. VMTS contains an OCL 2.0 compliant constraint compiler that generates a binary executable for constraint validation [7]. The constraints contained both by the rewriting rules and by metamodel diagrams are attached to the metamodel, thus they can be handled with the same algorithms.

Previous papers [8] and [9] have presented three optimization algorithms. These algorithms can reduce the navigation steps in the constraints (i) by relocating the constraints, (ii) separating clauses based on Boolean operands and (iii) caching the result of the model queries applied during validation. The main advantage of the algorithms is that they do not rely on system-specific features, thus, they can easily be implemented in any modeling or model transformation framework. The general correctness of the algorithms has also been proved.

While implementing and by further examining these algorithms, we have refined their application conditions. We have found that the scope of usability of the first algorithm is limited. Furthermore, the second algorithm can accelerate the validation in certain cases only, according to the type of the Boolean operand. The cases where the decomposition to clauses are meaningless, thus, the advantage of the optimization that equals zero have to be excluded from the algorithm. The primary aim of the paper is to present these restrictions and the extended algorithms.

The paper is organized as follows: firstly, Section 2 elaborates the original version of the two optimization algorithms. Secondly, Section 3 introduces the limitations of the algorithms, while Section 4 presents the new, extended algorithms. Finally, Section 5 summarizes the presented work.

2 Backgrounds and Related Work

In general, the evaluation of OCL constraints consists of two steps: (i) selecting the object and its properties that we need to check against the constraint and (ii) executing the validation method. Although the second step can use several OCL-related optimization methods, our optimization algorithms focus on the first step, bacause: (i) The efficiency of the validation depends on the realization of the OCL library (types and expressions), thus, optimizing the validation process is usually more implementation-specific; (ii) in general, the first step has more serious computational complexity, since each navigation step means a query in the underlying model. The original version of the algorithms were published in [8] and in [9].

2.1 Relocation

One of the most efficient way to accelerate the constraint evaluation is to reduce the navigation steps in a constraint. This is the aim of the first algorithm,

called *RelocateConstraint* (Alg. 1). The algorithm processes the propagated OCL constraints, and tries to find the optimal context for the constraint. The main *foreach* loop examines the navigation paths of the actual constraint and relocates the constraint to the node at the smallest navigation cost. Here, relocation means changing the context of the constraint without changing the result of the evaluation.

Algorithm 1 RelocateConstraint algorithm

```
1: RelocateConstraint(Model M)
2: for all InvariantConstraint C in M do
3:    minNumberOfSteps = CalculateSteps(CurrentNode in C)
4:    optimalNode = CurrentNode of the C
5:    for all Node N in C do
6:       numberOfSteps = CalculateSteps(N)
7:       if numberOfSteps < minNumberOfSteps then
8:          minNumberOfSteps = numberOfSteps
9:          optimalNode = N
10:   if optimalNode ≠ CurrentNode of C then
11:      UpdateNavigations of C
12:      Relocate C to optimalNode
```

2.2 Decomposition

Constraints are often built from sub-terms and linked with operators ($self.age = 18$ *and* $self.name = 'Jay'$), or require property values from different nodes ($self.age = self.teacher.age$). Thus, using the *RelocateConstraint* algorithm, it is not always possible to eliminate all navigation steps. Although these subterms are not decomposable in general, they can be partitioned to clauses if they are linked with Boolean operators. A clause can contain two expressions (OCL expression, or other clauses) and one operation (AND/OR/XOR/IMPLIES) between them. By separating the clauses, we can reduce the number of the navigation steps contained by the OCL expressions and the complexity of the constraint evaluation during the constraint validation process. It is simpler to evaluate the logical operations between the members of a clause than to traverse the navigation paths contained by the constraints.

The ANALYZECLAUSES algorithm (Algorithm 2) works on the syntax tree of the constraint. The algorithm is invoked for the outermost OCL expression of each invariant, recursively searches the constraint for possible clause expressions and creates the clauses. The algorithm uses the following rules: (i) A clause is created for every logical expression, the two sides of the expression are added to the clause as children. The children are recursively checked to decompose nested Boolean relations. (ii) Parentheses are eliminated, the inner expressions are checked. (iii) In other cases, if there is only one expression in the whole

constraint, then a special clause is created, otherwise the *RelocateConstraint* algorithm is used on the expression.

Algorithm 2 AnalyzeClauses algorithm

```
 1: AnalyzeClauses(Model Exp)
 2: if Exp is LogicalExpresssion then
 3:    Clause = CreateClause(Exp.RelationType)
 4:    Clause.AddExpression(AnalyzeClauses(Exp.Operand1))
 5:    Clause.AddExpression(AnalyzeClauses(Exp.Operand2))
 6:    return Clause
 7: else
 8:    if Exp is ExpressionInParentheses then
 9:       return AnalyzeClauses(Exp.InnerExpression)
10:    else
11:       if Exp is OnlyExpressionInConstraint then
12:          Clause = CreateClause(SpecialClause)
13:          Clause.AddExpression(RelocateConstraint(Exp))
14:          return Clause
15:       else
16:          return RelocateConstraint(Exp)
```

3 Contributions

In general, there are two key questions in connection with optimization algorithms: (i) whether they result in the same output as the original algorithm for every possible input and (ii) whether they are more efficient. The first question is crucial, because having proper evaluation results is essential. These guidelines, these two questions are taken into examination when constructing the limitations for the optimization algorithms.

3.1 Correctness

Primarily, the correctness of the relocation algorithm is taken into examination. An algorithm or a relocation is *correct* only if the output of the optimized and original constraint is the same for every possible input. The aim of the limitations is to eliminate the cases where the result of the original and the optimized algorithms would differ. To achieve this, it is necessary to examine when and how *correct* relocations can be applied. In the following propositions, we often say — for the sake of simplicity — that a *RelocationPath* is *correct*, although we mean that the relocation using the *RelocationPath* is correct.

Proposition 1. *If the steps of* RelocationPath *are separately correct, then their composition, the* RelocationPath *is also correct.*

Example 1. The original constraint is located in node A, the optimal node is D (Fig. 1). Thus, the *RelocationPath* is drawn from A to D (dashed line). If neither the relocation from node A to C (solid line), nor the relocation from node C to D (dotted line) change the result of the constraint, namely they are *correct*, then the proposition states that the relocation from A to D is also *correct*.

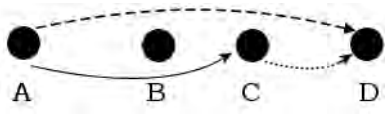


Fig. 1. The steps and the whole RelocationPath

Proof. Let C be the original constraint and P a complex *RelocationPath* found by the search steps. P contains finite number of steps, since the host model contains finite number of model items and no circular navigation paths are allowed. Furthermore, let O be the original context; S the first step of P and O' the destination node of S in P. According to the premise of the proposition the correctness of S is proven, thus, relocating the constraint from O to O' can be accomplished. After applying this relocation, a new constraint, C' can be constructed. Applying the relocation algorithm on C' results a new *RelocationPath*, P' containing one less step, than the original one. Since P has a finite number of steps, the algorithm always terminates.

Corollary 1. *The steps in a path can be examined separately. If in a certain case the correctness of the algorithm is proven to be* correct *for each single navigation step in the* RelocationPath*, then it is also proven for the whole* RelocationPath*. Thus, in general, if the correctness of each possible single navigation step is proven, then the correctness of the whole relocation is proven. Therefore, it is enough to examine the correctness of single relocation steps.*

In the next propositions, the following abbreviations are used: C denotes the original constraint, C' the new constraint, M_0 is metamodel, M is model, O is the original context, N is the new context. O and N are metamodel elements, and their instantiations are $O_1,\ O_2 \ldots O_n$, and $N_1,\ N_2 \ldots N_n$.

Example 2. Fig. 2 shows an example metamodel, its instantiation, and the constraint relocation. The metamodel represents a domain that can model computers and display devices (here monitors only). A single computer can use multiple monitors. The model defines a simple constraint attached to the node *Computer*, this constraint is relocated by the optimization to the node *Monitor*.

Using the abbreviations, we can say the following: M_0 is the metamodel shown in Fig. 2/a, M is its instantiation (Fig. 2/b). O is Computer, N is Monitor in M_0. O has two instantiations, Computer1 (O_1) and Computer2 (O_2).

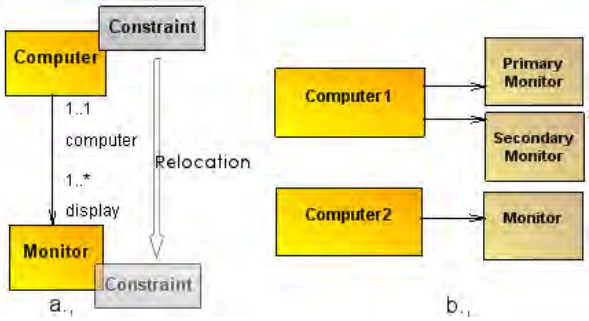


Fig. 2. Example metamodel and model

Similarly, PrimaryMonitor is N_1, SecondaryMonitor is N_2, and finally, Monitor is N_3.

Proposition 2. *Navigation edges that allow zero multiplicity (on either or both sides) cannot be used in* RelocationPath*.*

Proof. Let M be a model with O_1, N_1 *and* N_2 defined (Fig. 3). Let N_1 be isolated (or at least not connected with O_1).

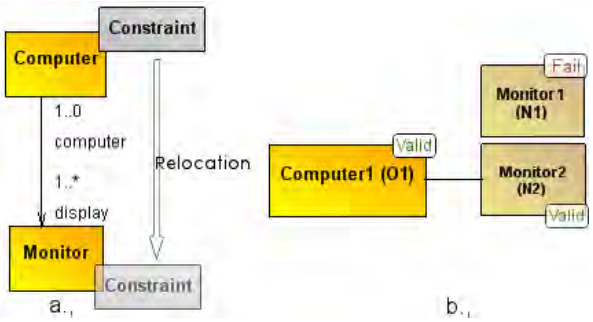


Fig. 3. Null multiplicity - metamodel and model

Let C and thus C' contain an expression that is not valid in N_1, but valid in N_2. The evaluation of C results true, since N_1 is not checked, because it is not connected with O_1. However C' fails, thus, the relocation is not *correct*.

The multiplicity of relations in metamodels is defined by a lower, and an upper limit. The limits can contain an integer representing the number of participants exactly, or * allowing any number of objects. In the following propositions, we categorize the multiplicities:

– *ZeroOrMore* - the lower limit of the multiplicity is 0 (the upper limit is not important)
– *ExactlyOne* - the lower and the upper limit is also 1
– *MoreThanOne* - the lower limit is not 0, while the upper limit is more, than 1

Proposition 3. *A relation with multiplicity ExactlyOne on both sides can be used for relocation. In this case the relocated expression differs from the original version in the navigation steps (or navigation step sequences). The new constraint expression is transformed from the original definition using the following rules:*

Rule 1. *If the expression is a navigation to the new context (N), then the expression is transformed into* `self`.

Rule 2. *If the expression is an attribute query in the old context (O), then the new expression is a navigation from N to O and an attribute query applied there (e.g.* `self.Manufacturer` *is transformed to* `self.computer.Manufacturer`*).*

Rule 3. *If the expression is a navigation from the old context (O), then the new expression is a navigation from N to O.*

Rule 4. *Other expressions in the constraint are not altered.*

Example 3. Let the example metamodel cited above define that computers are able to handle exactly one monitor, and monitors are always connected to exactly one computer (Fig. 4). Furthermore, let the constraint C state that the monitor is an LCD monitor ($display.Type = \ 'LCD'$). In this case relocating the constraint will result C': $Type = \ 'LCD'$.

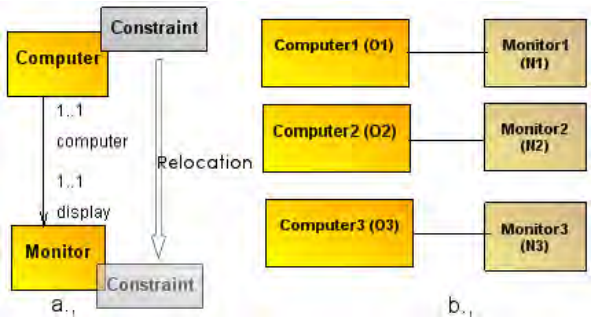


Fig. 4. ExactlyOne multiplicity on both sides - metamodel and model

Proof. An ExactlyOne multiplicity on both sides means that O and N objects can refer to each other the same way (using the role name of the destination node). The result of the navigation reference is always a single model item, not

a set of model items and not an undefined value. This means that changing the navigation steps can be accomplished.

The transformation rules are also correct if the rules above are satisfied:

Rule 1. The relocation has changed the context, thus, the navigation step in the original context is not necessary any more.

Rule 2. and **Rule 3.** Since the original attribute reference, or the destination node of the navigation is invalid in the new context, thus, the constraint has to navigate back to the original context first, and applying the expression there.

Rule 4. Rule 1-3. covers all possible valid attribute and navigation expressions, thus, no additional rules are required.

Proposition 4. *If the multiplicity is ExactlyOne on the destination side, but MoreThanOne on the source side (not allowing zero multiplicity), then the constraint expression can be always relocated. In this case the constraint is encapsulated by a new constructed* `forall` *expression. If the relocated constraint does not contain any attribute reference to the original context node, or navigation through it, then the* `forall` *expression can be avoided.*

The original expression cannot be used after relocation, because of the multiplicity MoreThanOne, which retrieves a set of model items. The basic idea is to create an iteration on the elements of the set; the iteration is not contained in the original constraint.

Example 4. Let O contain a simple constraint referring to one of its attributes, named `IsAbstract`. After the relocation, the constraint is located in N and the reference `self.IsAbstract` is transformed to

```
self.O->forall(O | O.IsAbstract).
```

This `forall` expression is true only if the condition holds for every elements in the set.

Example 5. The example model has been changed to meet the requirements of the proposition (Fig. 5).

Let C be defined as `self.Price < display.Price`. If this constraint is relocated, then it is transformed to

```
self.computer->forall(computer| computer.Price > self.Price)
```

expressing that *each* computer attached to the monitor has to accomplish the condition. Note that the navigation from O to N in `display.Price` was reduced to a single `self` reference similarly to the ExactlyOne-ExactlyOne case.

Proof. The presented method ensures that each model item on the original source side is processed, and the constraint is checked for each model item. Since the ZeroOrMore multiplicity is not allowed, the navigation is always possible. Inside the `forall` loop, the name of the destination node is the iterator value. Thus, this solution simulates ExactlyOne multiplicity on both sides. The relocated and the original version are equivalent.

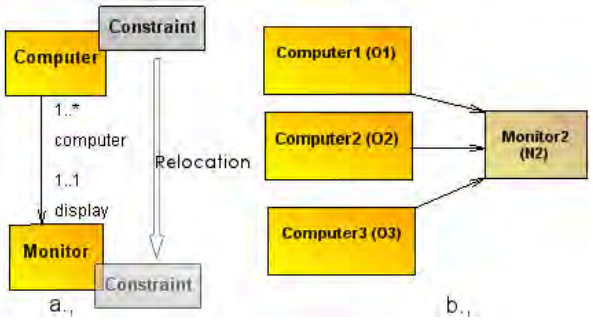


Fig. 5. MoreThanOne → ExactlyOne multiplicity - metamodel and model

Proposition 5. *If the multiplicity is ExactlyOne on the source side, but MoreThanOne on the destination side (not allowing optional multiplicity), then the constraint expression can be relocated if and only if the original expression uses* `forall`*, or* `not exists` *expression to obtain the referenced model items of the new context. This means that only those relations can be used where the original navigation selects all of the model items, or none of them (no partial selection, or another operation is allowed).*

Example 6. The constraint `self.N->count()` or `self.N->select(N.IsUnique)` cannot be relocated, but the constraint `self.N->forall(N.IsUnique)` can.

Example 7. The example model shows the requirements of the proposition (Fig. 6). Note that due to the preconditions of the proposition, the references to Monitor are always set operations in Computer. This means that, for example, the expression `self.display.Price>300` cannot be used, because `display` is a set, not a single value.

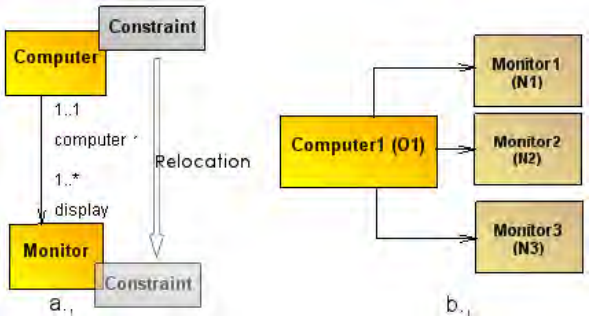


Fig. 6. ExactlyOne → MoreThanOne multiplicity - metamodel and model

Let M_0 contain three constraints: C_1, C_2 *and* C_3 using the following definitions:

```
inv c1: self.Price > 650
inv c2: self.display->count() > 5
inv c3: self.display->forall(m:Monitor| m.Price<300)
```

The proposition requires constraints to use `forall` expressions to query the attributes of the new context, or the navigation paths through the new context. But this also means that any other expression can be applied (for example a local attribute query, such as in *c1*). In this case the method of ExactlyOne-ExactlyOne multiplication can be used, thus, C_1' becomes the following:

```
inv c1: self.computer.Price > 650.
```

Complex set operations cannot be relocated according to the proposition, thus, C_2 cannot be relocated either. This limitation does not apply to C_3:

```
inv c3: self.Price<300.
```

Although the original and the relocated version of the constraint seems to differ, they have the same meaning: all monitors must be cheaper than 300 USD.

Proof. Firstly, the limitation to set operations is proven. In case of the general selection operations, such as `exists`, the selection criterion is *true* for some of the items and *false* for the others. This can lead to two problems with the constraint rewriting: (i) the constraint validation can generate false results where the selection criteria in the original expression is *true*/*false*, and (ii) the partial results arising in N cannot be processed (for example summarized) in O. Neither of these problems can be solved, thus, an universal relocation in this case is not possible.

Secondly, it needs to be proven that relocation is possible along `forall`, or `not exists` expressions. Note that `not exists` can be expressed using `forall` by negating the condition. The main difference between the previous (erroneous) subcase and this one is that here — if the model is valid — the condition in the select operation is *true* (or *false*) for *each* model item. Thus, the relocated constraint fails only, when the original constraint also fails. The relocation algorithm transforms `forall` expressions to single references. The relocated constraint is checked for each node of the new context, thus, the constraints are functionally equivalent.

Proposition 6. *If the multiplicity is MoreThanOne on both side (not allowing zero multiplicity) (Fig. 7), then the constraint expression can be relocated if and only if the original expression uses* `forall`*, or* `not exists` *expressions to query the referenced model items of the new context node.*

Proof. This case is a combination of the previous cases. A new `forall` expression is constructed such that it contains the whole relocated constraint, then, inside this newly constructed `forall`, the original `forall` and `not exists` expressions

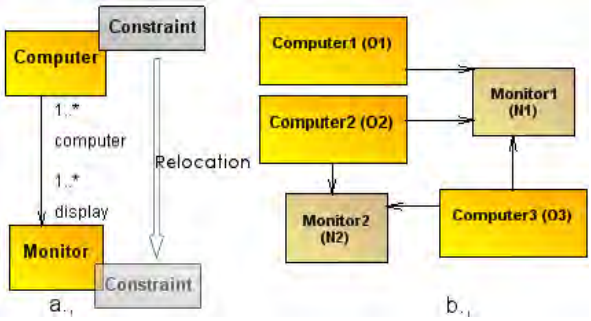


Fig. 7. MoreThanOne multiplicities - metamodel and model

are transformed to single navigation steps. The outer `forall` ensures that each O object is checked for each N, while the inner expression holds the transformed original constraint.

Proposition 7. *If the constraint contains more than one attribute reference expressions and these expressions do not depend on each other, then* partial relocation *is feasible.* Partial relocation *means that some of the expressions are executed in the new context, while others are executed in the original context. The original context is reached using navigation. Partial relocation does not apply to edges with zero multiplicity.*

Proof. Since the proposition is true only for relations not allowing zero multiplicity, the navigation between the original and the new context is always possible. Both ExactlyOne and MoreThanOne relations can be traversed according to the constructs presented earlier (either by single navigation steps, or `forall` expressions). Thus, when the constraint is evaluated, navigating back to the original context is always possible. In this way, the relocated and the original functionality is the same.

Corollary 2. *The task of finding possible destinations of relocation can be reduced to a simple path-finding problem from the original context to the new one, where relations allowing zero multiplicity cannot be the part of the path. Note that this path, if exists, is the* RelocationPath *mentioned earlier.*

One of the main difference between the *RelocateConstraint* and the *AnalyzeClauses* algorithm is that the first one modifies the constraint only by relocating it, but the algorithm does not need special support from the validation framework. In contrast, the second algorithm does not really modify the text of the constraint, but it requires support for clause-handling during validation. Therefore, *AnalyzeClauses* relies more on the framework, but depends less from the constraint text.

Proposition 8. *The original version (Algorithm 2) of the constraint decomposition algorithm (*AnalyzeClauses*) is always* correct *if the supporting functions are* correct*.*

Proof. The algorithm *AnalyzeClauses* consists of a condition and several method calls in the condition branches. The conditions ensures that the correct execution branch is selected in all cases. Thus, the only part of *AnalyzeClauses* that can cause erroneous results is the clause-handling, which is an external function. If this external function is implemented well, the decomposition is always correct.

3.2 Efficiency

RelocateConstraint can reduce the number of navigation steps in the constraint, but since the optimization uses only the metamodel, not the models, it does not know exactly how many model items are affected by a single navigation step. If the model uses ExactlyOne multiplicities only, then the optimization is *correct*, but the cost of navigation is not predictable if the model contains MoreThanOne multiplicities. In this case the number of model items on the destination side can vary, thus, for example, the algorithm cannot decide between two paths different only in relation with MoreThanOne multiplicities. This problem can be handled using heuristics, but a globally optimal method cannot be constructed.

The situation is completely different in case of *AnalyzeClauses*. Here the performance gained from the optimization depends on how efficient the construction of the clauses is. The basic idea behind the algorithm is that the result of the Boolean operations sometimes requires the evaluation of one of the operands only. For example in an AND expression, such as `self.Size>50 and self.display.Size>80` it is enough to check the value of the first operand if it evaluates to *false*. This is why the boolean operators are special, why the *AnalyzeClauses* algorithm is based on them instead of other types of operations.

Since the operands cannot affect each other, they can be evaluated separately according to [9]. In case of AND, OR and IMPLIES operations the value of one operand can affect the results of the whole operation:

- If either operand is *false*, then the AND operation is always *false*.
- If either operand is *true*, then the OR operation is always *true*.
- If the first operand is *false*, then the IMPLIES operation is always *true*.
- If the presented condition for the given operand is not satisfied, then both operands is evaluated.

Similar simplification is not available for XOR operations, because in this case both operands need to be evaluated.

4 The new algorithms

The restrictions to the optimization algorithms were presented in the previous section, now the last step is to construct new, extended algorithms according to these limitations.

The new *RelocateConstraint* is shown in Algorithm 3. It consists of two major parts: (i) searching for the optimal node (and RelocationPath) (Algorithm 4) and (ii) relocating the constraint if necessary (Algorithm 5).

Algorithm 3 The new RelocateConstraint algorithm

```
1: RelocateConstraint(Constraint, OriginalContext)
2: OptimalPath = SearchOptimalNode(OriginalContext, NULL)
3: if OptimalPath.LastElement ≠ OriginalContext then
4:    UpdateAndRelocate(Constraint,OptimalPath)
```

The first part of the *RelocateConstraint* algorithm is based on the *SearchOptimalNode* function. This function checks the relocation requirements while searching (*StepIsValid*), thus invalid*RelocationPath* candidates are dropped as soon as possible. *SearchOptimalNode* uses a recursive breadth-first-search strategy to find every possible candidates. The external function *CalculateSteps* calculates the number of model queries in the case when the new context is located in N.

Algorithm 4 The SearchOptimalNode algorithm

```
1: SearchOptimalNode(Node N, PathP)
2: minSteps = CalculateSteps(N)
3: optimumCandidate = Append(P, N)
4: for all CN in ConnectedNodes(N) do
5:    if StepIsValid(CN) then
6:       LocalOptimum = SearchOptimalNode(CN, Append(P, N))
7:       LocalSteps = CalculateSteps(LocalOptimum.LastElement)
8:       if LocalSteps < minSteps then
9:          minSteps = LocalSteps
10:         optimumCandidate = LocalOptimum
11: return optimumCandidate
```

The result of *SearchOptimalNode* is the *RelocationPath*. The last element of the path is the new context itself. If the new context and the old context are not the same, then the constraint is relocated and updated by the function *UpdateAndRelocate*. The relocation is based on path steps, thus, the algorithm updates the context declaration step-by-step. The multiplicity checking and the constraint updating mechanisms are implemented in external functions to improve the readability of the algorithm.

Algorithm 5 The UpdateAndRelocate algorithm

```
1: UpdateAndRelocate(Constraint C, Node O, Path P)
2: for all Step in P do
3:    if SourceMultiplicity(Step)= ExactlyOne and
      DestMultiplicity(Step)= ExactlyOne then
4:       ExactlyOneRewrite(C)
5:    if SourceMultiplicity(Step) ≠ MoreThanZero then
6:       AddForeach C)
7:    if DestMultiplicity(Step) ≠ MoreThanZero then
8:       RemoveForeach(C)
9: return optimumCandidate
```

In the case of *AnalyzeClauses* there is only one new limitation: *XOR* operations are excluded when creating the clauses. The algorithm is presented in Algorithm 6.

Algorithm 6 AnalyzeClauses algorithm

```
1: AnalyzeClauses(Model Exp)
2: if (Exp is AndExpression) or (Exp is OrExpression) or
   (Exp is ImpliesExpression) then
3:    Clause = CreateClause(Exp.RelationType)
4:    Clause.AddExpression(AnalyzeClauses(Exp.Operand1))
5:    Clause.AddExpression(AnalyzeClauses(Exp.Operand2))
6:    return Clause
7: else
8:    if Exp is ExpressionInParentheses then
9:       return AnalyzeClauses(Exp.InnerExpression)
10:   else
11:      if Exp is OnlyExpressionInConstraint then
12:         Clause = CreateClause(SpecialClause)
13:         Clause.AddExpression(RelocateConstraint(Exp))
14:         return Clause
15:      else
16:         return RelocateConstraint(Exp)
```

5 Conclusions

Due to the importance of constraints in modeling and model transformation, efficient validation methods are required. Previous work has presented three algorithms, which can accelerate the validation. This paper has examined the algorithms, especially the relocation algorithm *RelocateConstraint*. Based on the results, several necessary limitations and modifications have been introduced to the original algorithms. The statements have been illustrated by small examples

and their correctness has also been proved. More complex examples — focusing on the acceleration gained from the optimization — can be downloaded from [6]. According to the novel results, the algorithms have been updated.

As this paper has shown, proving the correctness of the algorithms precisely is hard to manage. A mathematical formalism could help, but the current formalism of OCL is based on set theory, which is hard to use in examination of dynamic behavior. Abstract State Machines offer a technique that has successfully been used in many similar domains as formalism. Such a formalism could prove the correctness of the algorithms applying formal semantics. Therefore, we are currently working on the formalism of the algorithms either using and extending the old formalism, or creating a new, ASM-based formalism.

Although the steps of the three optimization algorithms have been made more rigorous, processing the OCL constraints is not optimal. The decomposition and the normalization of atomic expressions have reduced the navigation steps to the minimum, and the caching algorithm has reduced the number of queries, but further research is required to extend the scope of the optimization algorithms and accelerate the process. The validation process can be optimized by rewriting the constraints and avoiding time consuming expressions, such as *AllInstances*.

6 Aknowledgements

The paper is established by the support of the National Office for Research and Technology (Hungary).

References

1. Lengyel L., Levendovszky, T., Charaf H. : Compiling and Validating OCL Constraints in Metamodeling Environments and Visual Model Compilers, IASTED 2004
2. MOF QVT Specification, http://www.omg.org/docs/ptc/05-11-01.pdf
3. Warmer, J. , Kleppe, A.: Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition, Addison Wesley, 2003
4. UML 2.0 Specification homepage, http://www.omg.org/uml/
5. Mezei, G. , Lengyel, L. , Levendovszky, T., Charaf, H. : Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality, INES, 2006
6. VMTS Web Site, http://vmts.aut.bme.hu
7. Mezei, G. , Levendovszky, T., Charaf, H. : Implementing an OCL 2.0 Compiler for Metamodeling Environments, 4th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence
8. Mezei, G. , Lengyel, L. , Levendovszky, T., Charaf, H. : Minimizing the Traversing Steps in the Code Generated by OCL 2.0 Compilers, WSEAS Transactions on Information Science and Applications, Issue 4, Volume 3, February 2006, ISSN 1109-0832, pp. 818-824.
9. Mezei, G. , Levendovszky, T., Charaf, H. : An Optimizing OCL Compiler for Metamodeling and Model Transformation Environments, Working Conference of Software Engineering, 2006 (accepted)

An MDA Framework Supporting OCL

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff

Information Security, ETH Zurich, 8092 Zurich, Switzerland
{brucker,doserj,bwolff}@inf.ethz.ch

Abstract We present an MDA framework, developed in the functional programming language SML, that tries to bridge the gap between formal software development and the needs of industrial software development, e.g., code generation. Overall, our toolchain provides support for software modeling using UML/OCL and guides the user from type-checking and model transformations to code generation and formal analysis of the UML/OCL model. We conclude with a report on our experiences in using a functional language for implementing MDA tools.

1 Introduction

Model-Driven Engineering refers to the systematic use of models as primary engineering artifacts throughout the development life-cycle of software systems. The instance of Model-Driven Engineering based on the UML and defined by the Object Management Group (OMG) is called model-driven architecture (MDA). In UML, various model elements like classes or state machines can be annotated by logical constraints using the Object Constraint Language (OCL); for this reason, UML can be used as a formal specification language with diagrammatic syntax.

For Model-Driven Engineering in general and MDA in particular, *technical support* ranging over several stages of the software development process—requirements analysis, design, code generation—is vital. This holds to an even larger extent if semantic information like formal specifications are processed. Thus, a technical framework is needed that provides an infrastructure for model elements annotated by OCL.

In this paper, we present such a framework, comprising a toolchain that guides the development process from modeling in a CASE tool to code-generation and formal verification. In particular, our framework consists of a type-checking component allowing to represent OCL in a structured format which can be imported into our model repository (su4sml). This model repository can serve as a basis for model transformations. Moreover, su4sml is the basis for a template-based code generator supporting code-generation for the UML core and state machines, enriched by OCL specifications and access control policies specified using SecureUML. Further, this model can be directly transformed into a (formal) model for the theorem proving environment HOL-OCL [4].

As a distinguishing feature, su4sml is developed in the functional programming language SML [11]. For this reason, implementers of model transformations can profit from several techniques that have proven to be of major importance

for symbolic computations occurring naturally in compiler construction or theorem proving: pattern matching allows for direct representations of rules to be performed during transformation, higher-order functions allow for the compact description of search- and replacement strategies, and having a strongly typed language helps to detect many errors at compile time.

We also present an implementation of one particular extension of our framework for UML/OCL: namely support for the UML-based language SecureUML [2]. SecureUML is designed to enrich the business logic of a system (represented by a class diagram or a statechart) with a concrete access control model for objects and operations. By a model transformation [3], class systems and operation specifications are transformed such that a combined model is generated, incorporating security and functional aspects. During the transformation, several proof obligations are generated, making explicit under which conditions the business logic of a system is not interfered by its security model. With the help of our framework, the combined model can be transformed to code, while the proof obligations making this transformation "correct" (in the sense of "no bad interference") can be proven by HOL-OCL. Thus, our framework can be seen as a first step towards a uniform framework supporting both semantic and code-generative aspects of UML/OCL specifications.

The Plan of the Paper. After a general overview of the framework, we present its main components: In section 3, we describe the implementation of our model repository, in section 4 we present a template-based code generator and in section 5 we describe the interface to HOL-OCL. Finally, we describe the SecureUML instance and discuss our experiences and observations.

2 Our Framework: An Overview

In this section, we give an overview of our framework and present an exemplary toolchain in which it can be used. As a prerequisite, we introduce the tools and technologies our framework is based on.

2.1 Background

SecureUML. SecureUML [2] is a security modeling language based on RBAC [12]. In particular, SecureUML supports notions of users, roles and permissions, as well as assignments between them: Users can be assigned to roles, and roles are assigned to specific permission. Users acquire permissions through the roles they are assigned to. Moreover, users are organized into a hierarchy of groups, and roles are organized into a role hierarchy. In addition to this RBAC model, permissions can be restricted by *Authorization Constraints* (expressed in OCL formulae), which have to hold to allow access. SecureUML is generic in the notion of protected actions that can be assigned to permissions. These are specified in a SecureUML *dialect*.

The Dresden OCL2 Toolkit. The software platform provided by the Dresden OCL2 Toolkit (`http://dresden-ocl.sf.net/`), written in Java, provides manifold support for OCL. Among other tools, a parser and type-checker for OCL is included. The toolkit is designed for modularity and flexibility. Thus, the Dresden OCL2 Toolkit is a good basis for building new OCL-based tools, either by integrating it into a CASE tool directly or by using it as a standalone tool leveraging the provided XMI import and export facilities. In our setting, we especially benefit from the XMI export, which includes the typed-checked OCL constraints as abstract syntax using an XML-based encoding.

HOL-OCL. HOL-OCL [4] (`http://www.brucker.ch/projects/hol-ocl/`) is an interactive proof environment for UML/OCL. Its mission is to give the term "object-oriented specification" a formal semantic foundation and to provide effective means to formally reason over object-oriented models. On the theoretical side, this is achieved by representing UML/OCL as a conservative, shallow embedding into the HOL instance of the interactive theorem prover Isabelle [8] while following the standard [9] as closely as possible; in particular, we prove that inheritance can be represented inside the typed λ-calculus with parametric polymorphism. As a consequence of conservativity with respect to HOL, we can guarantee the consistency of the semantic model. On the technical side, this is achieved by automated support for typed, extensible UML data models. Moreover, HOL-OCL provides several derived calculi for UML/OCL that allow for formal derivations establishing the validity of UML/OCL formulae. Some automated support for such proofs is also provided, albeit the achieved degree of automation is not yet satisfactory.

2.2 The Toolchain

Our framework is completed by a toolchain (see Figure 1) that consists of a UML CASE tool with an OCL type-checker for modeling software systems. The framework provides a model repository, model analyzers and various code generators.

We use the UML CASE tool ArgoUML (`http://argouml.tigris.org`) and combine it with the Dresden OCL2 Toolkit. The Dresden OCL Toolkit uses a specialized metamodel combining the UML 1.5 and the OCL 2.0 metamodel. This results in an upward compatible extension of the UML 1.5 metamodel: every UML 1.5 model is still a model of the combined metamodel. Models expressed in his specialized metamodel can be exported using the XMI export.

We also developed a Java-based transformation tool, *su2holocl*, on top of the Dresden OCL Toolkit which transforms a SecureUML model into a semantically identical pure UML/OCL model. This model transformation is explained in more detail elsewhere [3].

At time of writing, our SML-based framework comprises

1. an XMI import supporting the UML 1.4 and 1.5 meta-model (e.g., as used by ArgoUML) and also a metamodel combining UML 1.5 and OCL 2.0 (as used by the Dresden OCL2 Toolkit),

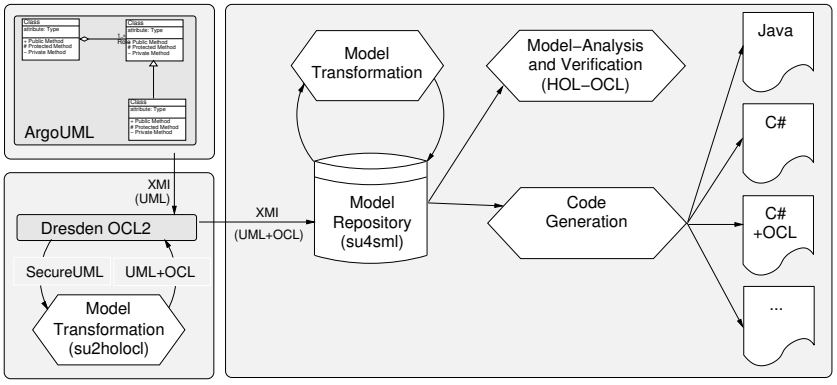


Figure 1. MDA Framework and Toolchain Overview

2. a model repository, *su4sml*, which supports the various metamodels we are using, e.g., UML, OCL, SecureUML,
3. a generic, template-based code generator supporting SecureUML (including the generation of access-control checks for the target languages Java and C#), the UML core (e.g., class diagrams), state machines, and OCL,
4. model transformations that normalize the models in several normal forms; this comprises the conversion of multiplicities into OCL constraints, etc., and
5. an interface to our theorem prover environment, HOL-OCL, which allows to do (formal) model analysis and verification of UML/OCL models.

The framework is implemented as a set of SML modules that are designed to be easily extensible and also can be used independently.

3 The Model Repository: su4sml

When implementing an object-oriented model repository in a functional programming language one has to solve several challenges: first one has to decide how to represent the inherently graph-based structure of object-oriented models into a tree-structure that is suitable in a functional programming language. Of course, one can always simulate pointers, but then one loses convenient features of functional programming languages, like safeness and strong typing.

For class models, we decided to employ the inherent tree structure given by the "containment hierarchy." For example, a class contains attributes, operations, or statemachines. We also decided to ignore associations as such. We only represent their association ends, again as part of the participating classifiers.

Statemachines, however, do not present an obvious way of representation in a tree structure. There we fall back to using pointers, for example from transitions to source and target states, or from states to incoming and outgoing transitions.

In contrast, OCL expressions naturally translate into an abstract datatype, as shown in Listing 1.1 and Listing 1.2. This abstract datatype is modeled closely

```
signature REP_OCL_TYPE = sig

  type Path = string list

  datatype OclType = Integer | Real | String | Boolean      (* Primitive Types *)
                   | OclAny | OclVoid
                   | Set of OclType | Sequence of OclType
                   | OrderedSet of OclType | Bag of OclType
                   | Collection of OclType
                   | Classifier of Path           (* user-defined classifiers *)
                   | DummyT              (* dummy type for untyped expressions *)
end
```

Listing 1.1. su4sml: Representing OCL Types

```
signature REP_OCL_TERM = sig
include REP_OCL_TYPE

datatype OclTerm =
     Literal            of string * OclType              (* Literal with type  *)
   | CollectionLiteral  of CollectionPart list * OclType (* content with type  *)
   | If                 of OclTerm * OclType             (* condition          *)
                         * OclTerm * OclType             (* then               *)
                         * OclTerm * OclType             (* else               *)
                         * OclType                       (* result type        *)
   | AssociationEndCall of OclTerm * OclType             (* source             *)
                         * Path                          (* assoc.-enc         *)
                         * OclType                       (* result type        *)
   | AttributeCall      of OclTerm * OclType             (* source             *)
                         * Path                          (* attribute          *)
                         * OclType                       (* result type        *)
   | OperationCall      of OclTerm * OclType             (* source             *)
                         * Path                          (* operation          *)
                         * (OclTerm * OclType) list      (* parameters         *)
                         * OclType                       (* result tupe        *)
   | OperationWithType  of OclTerm * OclType             (* source             *)
                         * string * OclType              (* type parameter     *)
                         * OclType                       (* result type        *)
   | Variable           of string * OclType              (* name with type     *)
   | Let                of string * OclType              (* variable           *)
                         * OclTerm * OclType             (* rhs                *)
                         * OclTerm * OclType             (* in                 *)
   | Iterate            of (string * OclType) list       (* iterator variables *)
                         * string * OclType * OclTerm    (* result variable    *)
                         * OclTerm * OclType             (* source             *)
                         * OclTerm * OclType             (* iterator body      *)
                         * OclType                       (* result type        *)
   | Iterator           of string                        (* name of iterator   *)
                         * (string * OclType) list       (* iterator variables *)
                         * OclTerm * OclType             (* source             *)
                         * OclTerm * OclType             (* iterator-body      *)
                         * OclType                       (* result type        *)
and CollectionPart = CollectionItem  of OclTerm * OclType (* element with type  *)
                   | CollectionRange of OclTerm           (* first              *)
                                      * OclTerm           (* last               *)
                                      * OclType           (* type of range      *)
end
```

Listing 1.2. su4sml: Representing OCL Expressions

```
signature REP_CORE = sig
type Scope
type Visibility
type operation =      { name          : string,
                        precondition  : (string option * OclTerm) list,
                        postcondition : (string option * OclTerm) list,
                        arguments     : (string * OclType) list,
                        result        : OclType,
                        isQuery       : bool,
                        scope         : Scope,
                        visibility    : Visibility }

type associationend = { name          : string,
                        aend_type     : OclType,
                        multiplicity  : (int * int) list,
                        ordered       : bool,
                        visibility    : Visibility,
                        init          : OclTerm option }

type attribute =      { name          : string,
                        attr_type     : OclType,
                        visibility    : Visibility,
                        scope         : Scope,
                        stereotypes   : string list,
                        init          : OclTerm option }

datatype Classifier = Class of { name        : Path,
                                 parent      : Path option,
                                 attributes  : attribute list,
                                 operations  : operation list,
                                 associationends : associationend list,
                                 invariant   : (string option * OclTerm) list,
                                 stereotypes : string list,
                                 interfaces  : Path list,
                                 activity_graphs : ActivityGraph list}
                    | Interface   of { ... }   (* similar to Class *)
                    | Enumeration of { ... }
                    | Primitive   of { ... }
end
```

Listing 1.3. su4sml: Representing the UML Core

following the standard OCL 2.0 metamodel. Note however, that OCL expressions include a lot of type information in this model. In essence, the type of each subexpression appears twice: once as the type of the subexpression itself, and once as the type expected (or inferred) as part of the enclosing expression. This constructions allows us, for example, to insert explicit typecasts that are only implicit in the original expression.

In addition to these datatype definitions, the repository structure defines a couple of normalization functions, for example for converting association ends into attributes with corresponding type, together with an invariant expressing the cardinality constraint.

Summarizing, the top-level data structures (see Listing 1.1, Listing 1.2 and Listing 1.3) of su4sml are inspired by the metamodels of OCL [9, Chapter 8] and UML [10] and readers familiar with these metamodels should recognize the similarities.

```
@// Example template for Java
@foreach classifier_list
        @openfile generated/$classifier_name$.java
        package $classifier_package$;

        public class $classifier_name$
        @if hasParent
                extends $classifier_parent$
        @end
        {
        @foreach attribute_list
                public $attribute_type$ $attribute_name$ ;
        @end
        @foreach operation_list
                public $operation_result_type$ $operation_name$(
                @foreach argument_list
                        $argument_type$ $argument_name$
                @end )
                {}
        @end
        }
@end
```

Listing 1.4. A Simplified Template File

4 A Template-Based Code Generator

We developed a *Generic Template-driven Code Generator* (GCG) on top of the su4sml repository. Template-based means that for each code artifact to be generated there is a template file which contains a skeleton of what has to be generated intertwined with instructions for the code-generator how to fill out the template. The code generator consists of a generic core and a set of cartridges that can be "plugged" into this core. The core part of GCG is independent both with respect to the input as well as the output language, the cartridges are responsible for interpreting the language-dependent instructions in the template files.

The template language has at the core just three syntactic elements: an @if statement for branching on Boolean predicates, a @foreach statement for iterating over lists, and $variable$ interpolation. The template language is not Turing-complete. For example, the predicates in @if statements come from a fixed (finite) set that is defined by the cartridges that are plugged into the core. Example predicates are attribute_isPublic or operation_isStatic. Similarly, the lists to iterate over are also defined by the cartridges. Example lists are classifier_list, attribute_list, or operation_list. These lists have an implicit notion of hierarchy. The attribute_list, for example, evaluates to the list of attributes of the current classifier that one iterates over in the enclosing @foreach statement. Finally, the variables that can be interpolated are also defined by the cartridges. Typical examples are operation_name or attribute_type, see Listing 1.4 for an example template file.

While the generic core parses the template file, the actual evaluation of the statements is delegated to the cartridges. For example, when the core executes the statement @if operation_isStatic, it asks the cartridge for the current value of

```
signature GCG = sig
  val generate  : Rep.Model → string → unit
end

functor GCG_Core (C: CARTRIDGE): GCG  = struct
    (* misc. auxiliary functions omitted *)

fun generate model template
  = let val env  = C.initEnv  model
        val tree = parse template
        in
            (initOut();
             write env tree;
             closeFile () )
             handle GCG_Error ⇒ (closeFile(); raise GCG_Error)
        end
end
```

Listing 1.5. GCG: the generic code generator

the predicate operation_isStatic. Depending on the answer, the core executes the following statements or not.

On the implementation level, the core is a functor which takes a cartridge as an argument (see Listing 1.5). The functor GCG_Core only takes one cartridge as an argument, whereas we want to be able to plug arbitrarily many cartridges together (see Figure 2). We achieve this by letting each cartridge be a functor itself, which takes another cartridge as an argument. In this way, we can build up cartridge chains supporting increasing functionalities. If one cartridge does not support a requested functionality, it passes the request on to the next cartridge, and the result back to the requester. To bootstrap this cartridge chain, we start with a cartridge that is not a functor. This could for example be a trivial cartridge that simply does nothing. For convenience, however, we implemented a base cartridge that implements the most basic functionalities which one would probably need in most languages anyways, for example, variables like attribute_name of lists like operation_list. The design allows for cartridges to override these functionalities by implementing them themselves. This is sometimes necessary for language-specific cartridges when the language requires certain syntactic properties. We implemented cartridges for Java and C# in this way.

To get a Java code generator, for example, one has to plug the cartridges together like follows:

```
structure Java_Gcg = GCG_Core (Java_Cartridge(Base_Cartridge));
```

The functor GCG_Core is applied over the cartridge resulting from the application of the functor Java_Cartridge over the base cartridge. The resulting structure Java_Gcg implements the signature GCG and therefore has a function generate which generates Java code from a given UML-model and a given template.

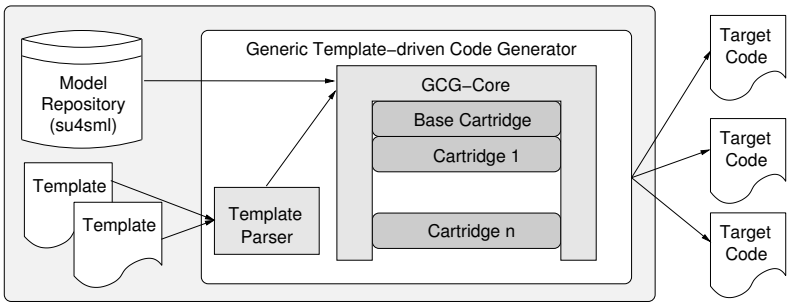


Figure 2. The Cartridge Chain Architecture

```
signature REP_ENCODER = sig
type mdr   = {   theory        : theory,
                 universe      : typ,
                 classifiers   : Classifier list }
  val add_classifiers : Classifier list → mdr → mdr
end
```

Listing 1.6. The Top-level Interface of the Repository Encoder

5 A su4sml-based Datatype Package for HOL-OCL

In this section, we present one vital component of HOL-OCL concerned with the encoding of object-oriented data structures in HOL, which is a tedious and error-prone activity to be automated. In this section, we give an overview of the su4sml-based datatype package we implemented to automate this process. In the theorem prover community, a *datatype package* [7] is a module that allows one to introduce new datatypes and automatically derive certain properties over them. A (conservative) datatype package has two main tasks:

1. generate all required (conservative) constant definitions, and
2. prove as much (interesting) properties over the generated definitions as possible automatically behind the scenes.

Our datatype package uses the possibility to build SML programs performing symbolic computations over formulae in a logically safe way over derived rules.

In the following, we give a brief overview what our package does ([4,5] describes more details). The datatype package is implemented on top of the su4sml interface on one hand and on top of the Isabelle core on the other (see Listing 1.6 for details). During the encoding, our datatype packages extends the given theory by a HOL-OCL-representation of the given UML/OCL model. This is done in an extensible way, i.e., classes can be added later on to an existing theory preserving all proven properties ([5] presents for more details). The obvious tasks of the datatype package are:

1. declare HOL types for the classifiers of the model,
2. encode the core data model into HOL, and
3. encode the OCL specification and combine it with the core data model.

```
fun cast_class_id class parent thy = let
  val pname = name_of parent
  val cname = name_of class
  val thmname = "cast_"^(cname)^"_id"
  val goal_i = mkGoal_cterm
             (Const(is_class_of class,dummyT)$Free("obj",dummyT))
             (Const("op␣=",dummyT)$(Const(parent2class_of class pname,dummyT)
               $(Const(class2get_parent class pname,dummyT)$Free("obj",dummyT)))
                                 $(Free("obj",dummyT)))
  val thm = prove_goalw_cterm thy [] goal_i
            (λ p ⇒ [cut_facts_tac p 1,  (* proof script *)
                    asm_full_simp_tac
                       (HOL_ss addsimps
                              [o_def,
                               get_def thy (parent2class_of class pname),
                               get_def thy (class2get_parent
                               class pname )]) 1,
                    stac (get_thm thy (Name mk_get_parent)) 1,
                    asm_full_simp_tac (HOL_ss addsimps [
                          get_def thy (is_class_of class),
                          get_thm thy (Name ("is_"^pname^"_mk_"^(cname)))]) 1,
                    stac (get_thm thy (Name ("get_mk_"^(cname)^"_id"))) 1,
                    ALLGOALS(simp_tac (HOL_ss))])
in
  (fst(PureThy.add_thms [((thmname,thm),[])] (thy)))
end
```

Listing 1.7. Proving Cast and Re-Cast (simplified)

In fact, the most important task is probably not that obvious: The package has to generate formal proofs that the generated encoding of object-structures is a faithful representation of object-orientation (e.g., in the sense of the UML standard [10], or Java). These theorems have to be proven for each model during its encoding phase. Among many other properties, our package proves that for each pair of classes A and B where B is a generalizationof A the following fact:

$$\frac{\mathsf{self.oclIsType(B)}}{\mathsf{self.oclIsKind(A)}} \tag{1}$$

as well as the more complicated property:

$$\frac{\mathsf{self.oclIsDefined()} \quad \mathsf{self.oclIsType(B)}}{\begin{array}{c}\mathsf{self.oclAsType(A).oclAsType(B).oclIsDefined()}\\ \mathsf{and\ self.oclAsType(A).oclAsTypeB.oclIsType(B)}\end{array}} \tag{2}$$

Listing 1.7 presents a simplified version of the SML function cast_class_id that proves the property (2). The expression starting in line 5 generates a type-checked instance of the current theorem to prove with respect to the current class (and its parent). Readers familiar with LCF-style theorem provers will recognize the "proof script" in lines 10 to 23. Finally, the function registers the proven theorem in Isabelle's theorem database. Logical rules like (1) or (2) or co-induction schemes given by class invariants constitute the object-oriented datatype theory of a given class diagram and represent the basic weapon for

proofs over them, in particular verifications of UML/OCL specifications. Stating these rules could be achieved by adding axioms (i.e., unproven facts) during the encoding process, which is definitively easier to implement. Instead, our datatype package generates entirely conservative definitions and derives these rules from them; this also includes the definition of recursive class invariants, which are in itself not conservative ([4] describes this construction in detail).

This strategy, i.e., stating entirely conservative definitions and formally proving the datatype properties for them, ensures two very important properties:

1. our encoding fulfills the required properties, otherwise the proofs would fail, and
2. doing all definitions conservatively together with proving all properties ensures the consistency of our model (provided that HOL is consistent and Isabelle/HOL is a correct implementation).

One might ask what benefit an end-user will get from conservativity after all. Its need becomes apparent when considering recursive object structures or recursive class invariants. Stating recursive predicates as *axiom* results in *logical inconsistency* in general. For example:

```
context A inv: not self.oclIsType(A)
```

This invariant requires for all instances of type A not to be of type A. Thus, it is in fact possible to state a variant of Russell's paradox which is known to introduce logical inconsistency in naive set theory. Inconsistency means that the OCL logic can derive any fact; this might be exploited by an automated tactic accidentally. Logical inconsistency is different from an *unsatisfiable* class invariant meaning "there is no instance." In particular, in an inconsistent system, each class invariant can be proven both satisfiable and unsatisfiable.

Our conservative construction requires proofs of side-conditions which will fail in paradoxical situations as the one discussed above (c.f. [4] for details) while admitting the "useful" forms of recursion in class invariants. To get an idea for the amount of work needed, the import of the "Company" model (including the OCL specification) presented in the OCL standard [9, Chapter 7] generates 1147 conservative definitions and proven theorems, the larger "Royals and Loyals" model [13] model generates 2472 conservative definitions and proven theorems. The load process usually proceeds in reasonable times.

Using HOL-OCL (see Figure 3) one can formally prove certain properties of UML/OCL specifications. For SecureUML specifications one can generate security-related proof obligations that can be formally analyzed, the details how and which proof obligations are generated is described elsewhere [3]. An example for an important standard property of a class diagram is consistency (i.e., there is at least one system state fulfilling all invariants, and there exist functions for all operation specifications satisfying the pre- and postconditions for legal states) of a model. Another important property is the refinement relation (e.g., forward-simulation [14]) between two class diagrams, stating that one model is a refinement of the other. A further interesting formal technique allows for proving that an implementation (i.e., a "method" in UML terminology) is compliant to

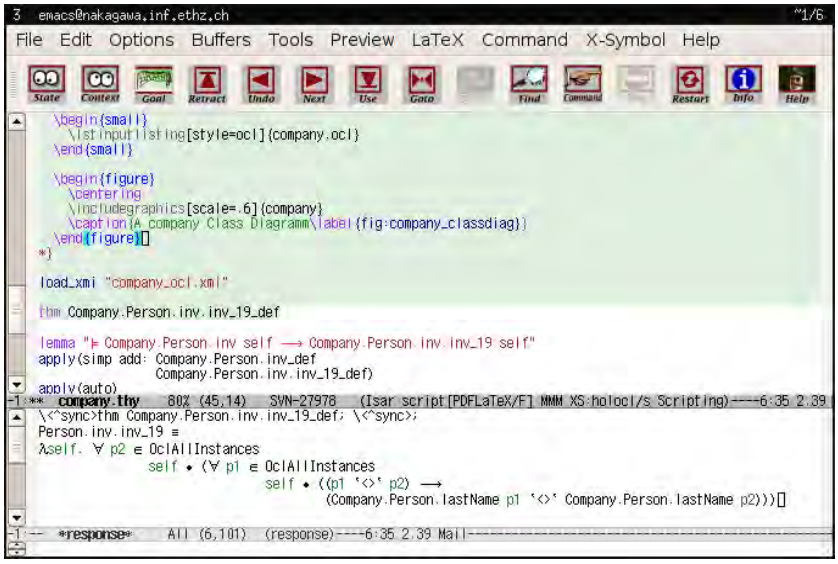


Figure 3. A HOL-OCL session Using the Isar Interface of Isabelle

a specification (i.e., a pair of pre- and postconditions). An in-depth discussion of these issues is out of the scope of this paper; with respect to the compliance problem, the reader might consult [5].

6 SecureUML Support

As we want to not only support standard UML/OCL models in our framework, but also SecureUML models, we have to extend the framework accordingly. We describe these extensions in the following sections.

6.1 SecureUML Support in the Model Repository

First, we have to extend the model repository to also contain model information coming from a SecureUML dialect.

```
signature REP_SECURE = sig
    structure Security : SECURITY_LANGUAGE
    type Model = Classifier list * Security.Configuration
    val readXMI: string → Model
end
```

This means, a "secure" model not only contains a list of classifiers (like the unsecured model), but also a security "configuration." The type of this configuration is parametrized by the concrete security language.

```
signature SECURITY_LANGUAGE = sig
    structure Design : DESIGN_LANGUAGE

    type Configuration
    eqtype Permission

    val getPermissions : Configuration → Permission list

    (* misc. auxiliary functions omitted *)

    val parse: Classifier list → (Classifier list * Configuration)
end
```

We currently have only one implementation of this signature, corresponding to the SecureUML metamodel, i.e., the permissions are given in terms of RBAC with additional authorization constraints. This design allows for other security languages, for example, a hypothetical PrivacyUML language. The function parse is responsible for extracting the security model information from a UML/OCL model, where it is usually given by a custom UML profile, i.e., stereotypes and tagged values.

The security language is itself parametrized by a design language, i.e., by a concrete SecureUML dialect.

```
signature DESIGN_LANGUAGE = sig
    eqtype Resource
    datatype Action = SimpleAction of string * Resource
                    | CompositeAction of string * Resource

    (* The resource hierarchy *)
    val contained_resources : Resource → Resource list

    (* the action hierarchy *)
    val subordinated_actions:   Action → Action list

    (* misc. auxiliary functions omitted *)

    val parse_action: Classifier → attribute → Action
end
```

The dialect specifies the actual resources and actions that are possible on these resources, together with the corresponding hierarchies over them. We implemented this signature both for the ComponentUML as well as for the ControllerUML dialect of SecureUML. Note the function parse_action, which is responsible for parsing the attributes of permission classes.

6.2 SecureUML Support in the Code Generator

After the repository has been extended, for code generation purposes we only need to define a corresponding cartridge. Implementing a cartridge mainly consists of deciding which "features" to support in the template language, i.e., which Boolean predicates, which lists, and which variables. As parts of this strongly depend on the SecureUML dialect, we implemented a SecureUML cartridge that again is parametrized by a SecureUML dialect. The SecureUML cartridge only knows about the global list of permissions, their assigned roles and constraints, which is information that is independent from the used dialect. The dialect specific cartridges then, e.g., deal with the assignment of actions to permissions.

6.3 SecureUML Support for HOL-OCL

At present, our datatype package for HOL-OCL supports SecureUML only indirectly using an external model transformation, su2holocl [3]. This model transformation converts a given SecureUML model into a semantically equivalent pure UML/OCL model. For the future, first-class SecureUML support for HOL-OCL is planned. The development of this support requires:

- the development of a machine-readable, formal semantics for SecureUML, e.g., as an embedding into HOL-OCL. Similar to the already existing theories covering the UML core and OCL, we have to develop a set of theories covering the SecureUML entities and their properties. For example, the development of a generic theory summarizing role-based access control models.
- the extension of the existing datatype package with support for the new SecureUML theories, i.e., the package must be extended to generate definitions for SecureUML entities and, if possible, the generation of security related proof obligations, together with proof attempts.

7 Conclusion

We have presented a framework for MDA comprising OCL support in model transformations, code generation and verification, together with one application of such a combined framework, namely SecureUML. In a way, our work can be seen as an approach to extend MDA with model-driven formal reasoning.

The code generator is a template-based generator which can be easily configured to produce code for various parts of models, target languages and target runtime-environments. The technique in itself is by no means new, but having it integrated into our framework and having access to structured OCL will, in our view, pave the way for new and up to now unexpected applications.

7.1 Related Work

Since code generation is at the heart of model-driven engineering, there is a wealth of similar approaches, e.g., AndroMDA (`http://www.andromda.org/`), which itself is based on Velocity (`http://jakarta.apache.org/velocity/`). Besides the fact that we apply functional programming techniques, there are two main differences: first, Velocity provides a rich template language with (among others) support for arithmetical, relational and logical operators over user-definable variables. Instead, our template language is intentionally very simple and restricted, but provides an `@eval` construct allowing for the execution of arbitrary SML code. The second difference lies in our concept of cartridges. Since a fixed, static template language is not flexible enough for generic code-generation, a template engine has to provide some support for extensibility. Velocity supports this by customizing and unstructured merging of the "context" object(s). In contrast, our concept of cartridges supports a notion of hierarchy and dependency between cartridges, which is type-checked on the SML module level. Our

cartridges also do not entail the complexity and overhead of AndroMDA cartridges, which include not only the template vocabulary, but also model-facades for the UML profile, and the template files themselves. Keeping these separated both simplifies the development of new cartridges and proves to be more flexible.

There are also some proof environments for OCL; since we focus on tool aspects and integration into MDA in this paper, we only mention the KeY Tool [1]. It offers a concrete verification method for a Java-like language (which HOL-OCL does not at present) at the dispense of compliance to the semantic foundations of OCL—the underlying semantics is a two-valued dynamic logic with an axiomatic representation of the data-models resulting from class diagrams.

With UMLsec [6] we share the conviction that security models should be integrated into the software engineering development process by using UML. However, although UMLsec provides a formal semantics, it does only provide rudimentary tool support, both for code generation and for (formal) model analysis.

7.2 Lessons Learned

Using Functional Programming Languages. Using a functional programming language for an object-oriented data model (e.g., the UML meta model) has advantages and disadvantages: on the one hand, a direct compilation into SML datatypes, i.e., mapping classes (with attributes) to constructors over records (with corresponding fields), leads to a quite substantial duplication of code for the inherited attributes and possibly in the pattern matching based functions processing these data structures. This representation of data models can be generated automatically from class diagrams via code generators such as our own (thus overcoming typical errors due to duplication). Nevertheless, pattern matching over constructors has to be designed and prepared with care to be extensible. For example, selector functions of inherited attributes like:

```
fun get_name (Class{name, ... })       = name
  |get_name (Interface{name, ...})     = name
  |get_name (Enumeration{name, ...}) = name
  |get_name (Primitive{name, ...})     = name
```

are sometimes preferable to pattern matching constructs since they are more stable under extensions; on the other hand, representing patterns only as selector and test functions, is feasible, but tedious and in itself very lengthy and error-prone. Thus, finding a suitable balance of re-usability and conciseness in each situation is the key for success.

We have been very pleased by the degree of abstraction and re-usability that has been achieved in the code generator by using the SML functor concept. To our knowledge, this is the first time that it had been applied to the concept of cartridges, which allows for a type-safe and aspect-oriented way to describe the compilation process. For example, the SML-structure containing the code generator for C# with SecureUML is constructed by the functor application:

```
1 structure CSharpSecure_Gcg
            = GCG_Core (SecureUML_Cartridge(CSharp_Cartridge(Base_Cartridge)),
                                            ComponentUML(Base_Cartridge));
```

which just represents it as a combination of the various compilation aspects.

Building a Toolchain. Our toolchain depends on a common XMI format for exchanging UML/OCL models. This has been the key for re-using work of other research groups in the field. However, in practice, each tool uses slightly different variants of the underlying meta-model, and thus different XMI variants. Full exchangeability of XMI files between different tools (and versions thereof) is still more a dream than reality. On the other hand, by having an infrastructure based on a general XML parser and pattern matching-based conversions between an imported XMI and the internal su4sml model repository, it turned out to be a fairly easy routine task to adapt to various XMI dialects. Such adaptions had been necessary several times during the lifetime of our project and could be realized usually in one day of programming work and turned out to be easier in practice than, developing and maintaining appropriate XSLT-transformations.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from UML models to access control infrastructures. ACM *Transactions on Software Engineering and Methodology*, 15(1), 2006.
3. A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, eds., *MoDELS 2006*, no. 4199 in LNCS, pp. 306–320. Springer, 2006.
4. A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zürich, 2006.
5. A. D. Brucker and B. Wolff. A package for extensible object-oriented data models with an application to IMP++ . In A. Roychoudhury and Z. Yang, eds., *SVV 2006*, Computing Research Repository (CoRR). 2006.
6. J. Jürjens. *Secure Systems Development with* UML . Springer, 2004.
7. T. F. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, eds., *The* HOL *Theorem Proving System and its Applications*, pp. 350–357. IEEE Computer Society Press, 1992.
8. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/*HOL*—A Proof Assistant for Higher-Order Logic*, LNCS , vol. 2283. Springer, 2002.
9. UML 2.0 OCL specification. 2003. `ptc/2003-10-14`.
10. OMG Unified Modeling Language Specification. 2003. `formal/03-03-01`.
11. L. C. Paulson. ML *for the Working Programmer*. Cambridge Press, 1996.
12. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
13. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for* MDA . Addison-Wesley, 2nd ed., 2003.
14. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

On OCL as part of the metamodeling framework MOFLON

C. Amelunxen, A. Schürr

Darmstadt University of Technology, Real-Time Systems Lab,
Merckstrasse 25, 64283 Darmstadt, Germany
[amelunx|schuerr]@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de

Abstract. The metamodeling framework MOFLON combines MOF 2.0, OCL 2.0 and graph transformations to generate sophisticated metamodel implementations. In this paper we describe the role of OCL in MOFLON. Furthermore, we present a set of constraints which corrects, completes and improves MOF 2.0 for the application as graph schema language.

1 Introduction

Nowadays, model driven software development is mostly realized by the application of the Unified Modeling Language (UML) [Obj05b]. Although UML satisfies many popular needs, sometimes domain specific languages are required to meet special concerns. Domain specific languages can be designed using the Meta Object Facility (MOF) [Obj06]. In its latest version 2.0, MOF offers constructs for the modeling of static structures, which in its original purpose should be used to describe the abstract syntax of modeling languages. Such a model of a modeling language (called metamodel) can be used to build an editor for the application of the modeled language. The datamodel of the editor can be generated from the language's metamodel [Dir02].

The generated metamodel reflects the abstract syntax of the modeled language which could only lead to a static datamodel. Additionally, the MOF compliant metamodel can be enriched by constraints formulated in the Object Constraint Language (OCL) [Obj05a]. From such a combination of MOF and OCL, metamodels with an additional constraint evaluation mechanism can be generated. The constraint evaluation offers a more precise verification of the static semantics and can be used to build an analysis component on top of an editor's datamodel.

The MOFLON framework [AKRS06] extends this approach by the application of story driven modeling [Zün01]. The combination of MOF, OCL and graph transformation allows the generation of very sophisticated metamodels which are far more than just a simple datamodel. Due to the specification of behavior by graph transformations, the generated metamodel can cover all actions that are based on the datamodel. Thus, the additional environment of the generated metamodel, for instance an editor GUI, may consist of just a tight, straight

forward implementation to instantiate the generated metamodel. Due to the fact that any logic related code is generated from a copious specification, we achieve a high degree of flexibility and maintainability. The combination of MOF, OCL and graph transformations offers the possibility to generate metamodels, which contain code to analyze metamodel instances as well as code to transform metamodel instances. The combination of analysis and transformation capabilities additionally provides the opportunity to specify transformations that correct metamodel instances in case of a failed analysis (by so-called repair actions), or in other words to transform the model if OCL constraints are violated.

Figure 1 gives an overview of the architecture of MOFLON. In the center of MOFLON, there is a MOF 2.0 metamodel which was created by a bootstrapping process. Beside MOF, OCL plays an important part in MOFLON. In the following we will concentrate on the relevance of OCL inside the MOFLON framework. Section 2 presents a scenario in which MOFLON and especially OCL as part of MOFLON can be applied in a very useful manner. Section 3 introduces a set of OCL constraints which we need to complete MOF 2.0 for the application as graph schema language. Finally we end up with a conclusion and a short overview about future work in section 4.

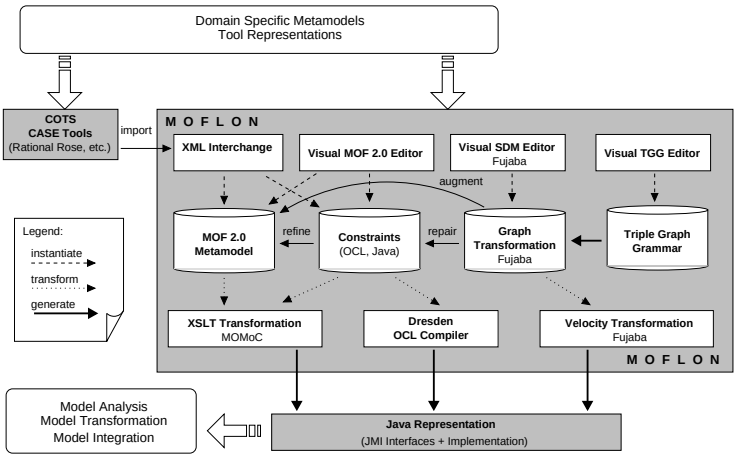


Fig. 1. Architecture of MOFLON

2 Application of MOFLON and OCL

One major problem of the model-based software and system developement is the surveilance of modeling guidelines which are indispensable in projects of a bigger size. The mere identification and formulation of design guidelines is a

problem which has to be handled manually without tool support. Opposed to that, the surveilance of the adherence to the formulated guidelines could be automated in the general case. In fact, the automation of the surveilance is in many cases mandatory just due the pure number and variety of the guidelines. Beside the automated surveilance and the automated identification and localization of modeling errors, the application of automated repair actions is a very desirable task [SDG+06]. As an example, we will demonstrate how MOFLON and especially OCL as part of MOFLON can help to generate code for the automated surveilance of design guidelines.

Figure 2 shows a very simple Matlab/Simulink model with a subsystem that is connected to a sink and a source through signals. The ports of the subsystem are named in adherence to a fictitious naming convention which demands that names of in-ports end with the suffix *_in*. First of all, the automatic surveilance of this modeling guideline requires a metamodel of Matlab/Simulink. A very simplified metamodel of Matlab/Simulink is depicted in Figure 3.

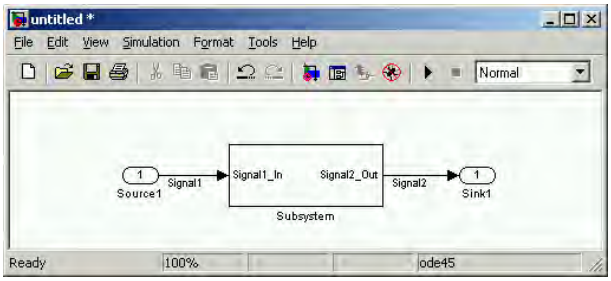


Fig. 2. Example of a design guideline in Matlab/Simulink

The metamodel covers only the elements which are necessary for the surveilance of the mentioned design guideline. *Blocks* can be connected with each other through *Ports* and *Signals*. A *Signal* connects exactly two ports, one in-port and one out-port. All these elements are named elements. At this point, OCL can be used in its original purpose to add precision by stating for instance that a *Port* can either refer to a *Signal* as *outSignal* or *inSignal*. Such a specification can only be used for code generation of static datamodels.

Additionally, MOFLON provides the feature to specify behavior using story driven modeling. Figure 4 shows the specification of the method *checkGuideline* which is intended to check if the mentioned design guideline is kept on a *Subsystem*. The behavior of the method is visually specified by a combination of activity diagrams and graph transformation rules. There is one graph transformation rule per activity (called story in this context). The transformation rule in the first story of Figure 4 matches all signals which are connected to an in-port of the subsystem the method is called on. Since only the ports which violate the mentioned design guideline should be handled by the transformation, the matching has to be controlled in such a way that only those ports are matched whose

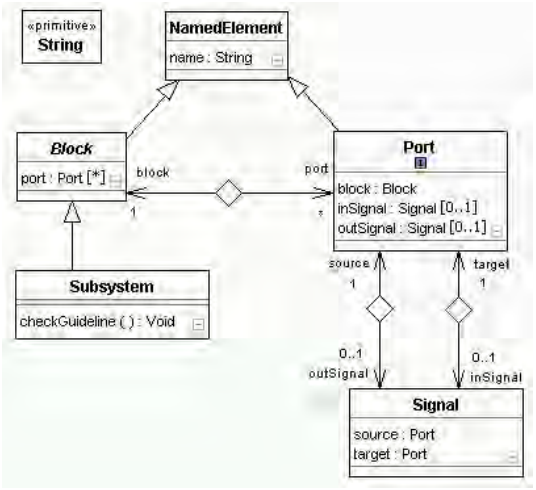


Fig. 3. Simplified metamodel of Matlab/Simulink

name do not end with the demanded suffix. At this point, we would also like to adopt OCL to be able to formulate constraints for the matching of attribute values which are more powerful than just a simple evaluation of attribute values. In general, OCL can also be used to determine the set of matched objects as a textual alternativ to complex graphical notations.

Considering the example, the violation of the design guideline is detected by an OCL constraint which checks whether the name of the port ends with the demanded suffix. In cases where such a match is found, the control flow of the activity diagram activates the second story in which an adequate repair action is formulated. An adequate repair action for the mentioned guideline is to set the name of the port to the name of its connected signal followed by the demanded suffix. Again, this manipulation[1] can be expressed by an OCL expression. The long term aim is to generate fully functional code for OCL constraints in the metamodel as well as for the OCL constraints in graph transformations. The outlined scenario is work in progress. Currently, MOFLON is able to generate JMI compliant metamodels enriched with code for the evaluation of invariants and code for the execution of SDM transformations.

Beside the integration of OCL into the matching and manipulation of the transformation rules and the application of OCL as constraint language for MOF, there is a third and very basic aspect how OCL is involved into the MOFLON approach. Since MOFLON applies MOF 2.0 as graph schema language, the static semantics of MOF are essential. They are the precondition for a proper application of OCL as mentioned before. Considering the graph transformation rules in Figure 4, the attribute *name* is used in the context of the class *Port*. An analysis which determines, if such a usage is possible has to be able to query all

[1] The manipulation of the attribute is indicated by the font color (green).

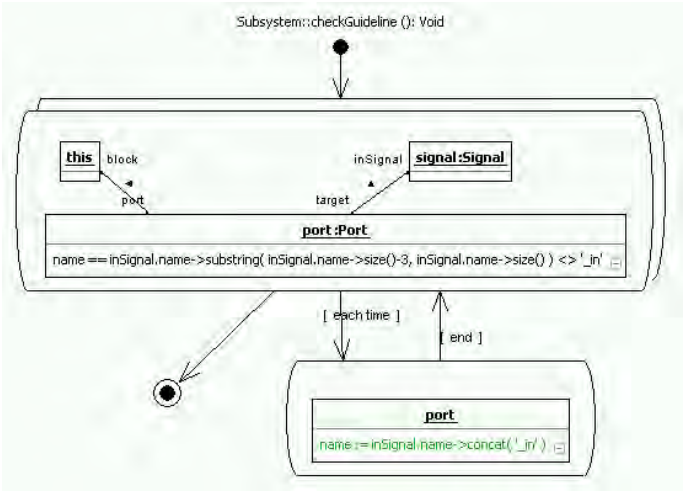


Fig. 4. OCL in graph transformations

inherited attributes of a class. The determination of inherited elements is part of the static semantics of MOF 2.0. In fact, the determination is formulated by an OCL constraint. Thus, the correct and complete static semantics are crucial for the complete integration of OCL in MOFLON. Therefore, in the following, this third aspect is described in detail.

3 Semantic completion of MOF 2.0 with OCL

As mentioned before in the context of Figure 1, MOF 2.0 is the central element of MOFLON. MOF 2.0 acts as metamodeling language for the static semantics of languages specified in MOFLON as well as schema language for the application of graph transformations. As such, the correct and precise semantics of MOF 2.0 are essential.

MOF 2.0 as the metametamodel of OMG's four-level metamodeling hierarchy is self describing. This means, that the static semantics of MOF 2.0 are basically described by MOF 2.0 metamodels but also additionally enhanced by OCL constraints and natural human language. The metamodels describing MOF 2.0 can be used to generate code for the basic features of MOF. Basic features comprise fundamental correlations like for instance the fact that associations are connected with classifiers through association ends. Such a correlation can be expressed by the metamodeling capabilities of MOF itself. But advanced and in many cases very important facts like the fact that only binary associations are allowed respectively in other words the number of association ends of one association has to be exactly two, can only be expressed with OCL.

The importance of OCL constraints can also be pointed out by considering the example of inheritance. The metamodel of MOF 2.0 only states that a classi-

fier can be generalized by a classifier. The fact that the generalization hierarchy has to be free of cycles can only be stated with OCL. Without the evaluation of OCL constraints, regardless of their implementation as hand written or fully generated code, a MOF 2.0 editor would be able to create arbitrary (e.g. cyclic) generalization dependencies. Although the importance of OCL constraints for the correct and complete static semantics of MOF 2.0 is often neglected, it has to attract careful attention for the purpose of applying MOF 2.0 as schema language for graph transformations.

Since MOFLON is developed by applying a bootstrapping process, the importance of a correct MOF 2.0 specification increases even more. We started our bootstrapping process with a simplified MOF 2.0 metamodel and a JMI compliant code generator (MOMoC [Bic04]). Based on the generated metamodel, we built an graphical editor and used this editor to improve the simplified MOF 2.0 metamodel of the editor. But even with the complete metamodel the editor does not prevent cyclic generalization dependencies, for instance. The bootstrapping process can only lead to an editor which reflects the complete static semantics of MOF 2.0 if the code generation also generates evaluation code for OCL constraints. Since, the metamodel consists of MOF and OCL, the bootstrapping can only be finished if both parts are reflected in the generated code. This can only be achieved if both parts are in a state which allows the application of a code generator. Hence, we had to analyze the MOF 2.0 specification for specification errors and impreciseness to be able to formulate a set of OCL constraints which formalizes MOF 2.0 up to a degree that is required for the application of graph transformations.

An application of a validation tool for syntax and type checking like done in [BGG04] for the UML 2 Superstructure might act as a basis. But since we are primarily interested in detecting impreciseness and specification leaks, an automated validation is not sufficient as imprecise semantics can be expressed even by proper and accurate constraints. Thus, we focused on a careful manual analysis to concentrate on semantical errors instead of detecting syntactical errors by the application of an OCL validater. Syntactical errors will at least be detected when we finish the bootstrapping process by the integration of generated evaluation code.

The result of our analysis is a set of OCL constraints which corrects and completes MOF 2.0 for the application as graph transformation language. The complete set of constraints cannot be presented in this paper. It is completely available at [Rea05]. In the following we provide information on the constraint set by introducing the error categories with some exemplary errors and the organization of the constraint set. The presented errors should provide an impression of the kind of errors that are part of the MOF 2.0 specification.

The presented examples refer to [Obj03] respectively [Obj04] where mentioned. In fact, both documents are not the latest available specifications, but since the combination of the presented corrections and improvements with the referred specification forms a complete and consistent specification a continuous adaption to the actual OMG documents does not seem reasonable to us. Never-

theless, a spot check of some selected errors leads to the result that some minor issues are fixed but major errors still exist.

3.1 Syntactical errors

The first and most obvious category of errors is the category of syntactical errors. Syntactical errors are more or less trivial to detect since they arise from a wrong usage of OCL. An automatic analysis would have been able to find such errors as well. In the following we will give some examples of typical syntactical errors in the MOF specification, apart from mere typos.

The cause of many errors is the wrong usage of OCL methods like, for instance, the application of the method *includes* with a collection passed as parameter instead of a single object (see [Obj04], p. 79). Another representative error is demonstrated by the application of squared brackets to access elements by their index: $forAll(i|op.ownedParameter[i].type.conformsTo(...))$ (see [Obj03], p. 149). Sometimes methods are used in a wrong context like the concatenation of strings by applying the method *union* (which should be applied to collections) instead of applying the method *concat*. Beside those obvious syntactical errors there are methods which are specified but never used (*bestVisibility*, see [Obj03], p. 93). Of course, this can hardly be considered as error, but at least it might confuse the reader of the specification.

3.2 Semantical errors

The category of semantical errors covers errors that are caused by any other reason than just the wrong application of OCL syntax. That may also comprise automatically detectable errors like wrong navigation in the metamodel. The navigation *association.owningAssociation* in the context of the class *Property* (see [Obj04], p. 131) for instance, is not possible. In fact, each of both association ends could be used to start navigation but a combination of both is not possible since both address the same class.

Another annoying but quite common kind of errors are wrong derivation rules. Some attributes of the metaclasses are derived by an OCL constraint. Those derived attributes are usually quite important, like for instance the derived set *inheritedMember*. It specifies all elements inherited from the general classifiers. The derivation rule for *inheritedMember* uses the specified method *hasVisibilityOf* which in turn also uses the attribute *inheritedMember* in its specification (see [Obj04], p. 85).

In fact, such an error would also be detected by an analysis tool whereas the following error is not that obvious and could only be detected by a human analysis. The derived attribute *importedMember* determines the named elements that are (transitively) imported from several namespaces into the importing namespace. Both kinds of import (package import and element import) have a visibility to determine whether the elements imported via a certain import will be exported from the importing namespace or not. In other words, the visibility of an import can be used to control the transitivity of the import. Thus, the

derivation rule for *importedMember* has to take the visibilities of the involved import relationships into account, which it does not (see [Obj03], p. 143).

There are also errors in form of specification leaks that arise from the complex package structure. The method *conformsTo* of the metaclass *Classifier* for instance is defined in the package *Generalizations*. The central package of the UML Infrastructure respectively MOF is the package *Constructs* which reuses several packages except the *Generalizations* package. Thus, the method *conformsTo* is not available in MOF although it is used. Beside the constraints that are wrong, the specification is also vitiated by bad namings and by missing constraints like for instance a constraint that prevents a namespace from importing itself.

3.3 Customizations and Improvements

A chance for customization and improvement does not necessarily require an error. On the one hand customizations and improvements contribute to a clear and consistent specification by clarifying complex and confusing constraints. On the other hand, additional constraints improve the completeness of the specification by handling special cases as well as optional and implicit demands. In fact, some actually correct constraints can be combined to a single constraint like for instance the two invariants for the determination of a named element's qualified name (see [Obj03], p.78).

Beside such trivial improvements there are also improvements with an important impact on the complete specification like exemplarily described in the following. As mentioned before, there are two kinds of import. First of all there is the commonly known package import that adds all the elements of the imported namespace with visibility set to *public* to the importing namespace. Second, there is the element import that adds only a single element with visibility set to *public* to the importing namespace. So obviously, the package import is just a shortcut notation for element imports on each element of the imported namespace and as such, both variants should be exchangeable (see [Obj03], p.145). But a problem arises due to the transitivity of the imports since an element import can use alias names for the imported element in the importing namespace. Thus, the constraints which determine the elements of a namespace must be able to query the alias name of an imported element. Without the loss of transitivity, this can only be achieved by extending the metaclass *ElementImport* by an attribute to determine previous element imports.

Indeed, a lot of modifications have to be made to take such extensive changes into account but without doing so, the specification would neither be correct nor consistent. Not all improvements contribute to the correctness of the specification. We propose also constraints whose adherence is not necessary but desirable. There are also situations that are implied by other situations. Such implications can also be expressed via OCL constraints. Therefore, we had to introduce several categories and levels of errors which are described in the following.

3.4 Categorization

The complete set of constraints we propose for a correct and consistent specification of MOF 2.0 can be found at [Rea05] in form of several tables. In the following we describe the classifications we made. The classifications are all reflected in the tables. All constraints are formulated in OCL 2.0.

First of all, the constraints are classified based on their error category. Additionally it is indicated whether the constraint is an existing constraint of the specification or an proposed extension. Table 1 summarizes the several categories.

Description	**Shortcut**	**Number**
Existing constraint is correct[2]	ok	1
Existing constraint ...[3]		
... is syntactical wrong.	SYN	2
... is semantical wrong or problematic.	SEM	3
... can or has to be customized.	CUS	4
Proposed constraint ...[4]		
... solves syntactical error.	SYN	5
... solves semantical errors or problems.	SEM	6
... improves or customizes.	CUS	7
... adds additional precision.	+SEM	8
... adds an additional customization.	+CUS	9

Table 1. Categorization of constraints regarding their kind of error.

As mentioned before, not every constraint has to be met for a correct metamodel instance. For some constraints, the adherence of the metamodel instances to the constraint is just desirable but not necessary. Therefore, we had to introduce a second categorization which is orthogonal to the categorization introduced first. We categorize the constraints into three different types of constraints. Table 2 introduces the three types of constraints.

Type of constraint	**Shortcut**
Mandatory for valid metamodel instances.	mand.
Optional for valid metamodel instances though useful	opt.
Implicit constraint	impl.

Table 2. Types of constraints regarding their role in the specification.

Constraints of type mand. have to be met for valid metamodel instances. If one single constraint of type mand. is broken the complete metamodel instance is invalid. Whereas constraints of type opt. do not contribute to the correctness

of the metamodel instances at all. A metamodel instance can be valid, although some or even all optional constraints are broken, as long as all mandatory constraints are met. But, the adherence to all optional constraints leads to useful metamodel instances without senseless constructs like for instance a namespace importing itself.

Beside these two quite obvious types, there is a third type i.e. the implicit constraints. An implicit constraint does not contribute to the correctness of metamodel instances as well. It indicates whether a metamodel instance contains constructs that are not explicitly drawn in the diagram but are implied by other constructs. For instance, considering the subsetting of association ends, the uniqueness of association ends which determines if multi-valued association ends may contain duplicates or not, is subject of implicit constraints. If one association end subsets another association end which is unique, it is implied that the subsetting association end is also unique. If it is mentioned in the diagram that the subsetting association end may contain duplicates, it indeed never will contain duplicates since the subsetted association end prevents such a situation by its uniqueness. Such implied correlations can be detected by OCL constraints.

Considering an editor which is build on top of a metamodel that is enriched by constraints of all three types, during evaluation, a violated mandatory constraint would rise an error message, a violated optional constraint would rise a warning message, whereas a violated implicit constraint would just cause an information message. That is how the constraint set will be integrated into MOFLON.

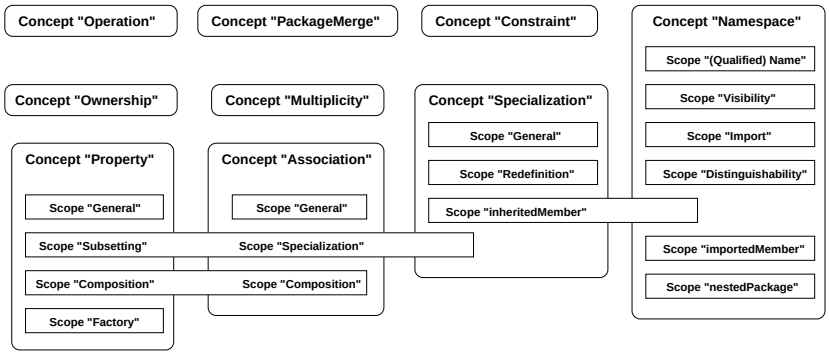


Fig. 5. Concepts and scopes of constraints

For a better structuring, the tables are separated according to their concept and the scope they cover. The constraints are grouped into concepts like for instance Association or Namespace. Those concepts do not map onto a package structure. They just group several constraints that contribute to the realization of the concept they are grouped into independent from their context. Constraints of the concept Association for instance are taken from the context

Core::Constructs::Association as well as from *Core::Constructs::Property*. Concepts are divided in several scopes to achieve a more flexible structure since some scopes contribute to more than just one concept. Figure 5 shows the complete set of concepts and scopes.

All in all, 90 constraints have been analyzed. As a result, 48 constraints (53%) have been considered free of any errors and 42 constraints (47%) were erroneous. Our constraint set provides 101 improvements in form of 86 OCL constraints. One half (51) of the improvements refer to existing constraints. The other half (50) improves the specification additionally. Fig. 6 shows the absolute numbers of constraints distributed over the categories of Table 1. The y-axis indicates the number of the category, the x-axis represents the number of occurrences in the proposed constraint set.

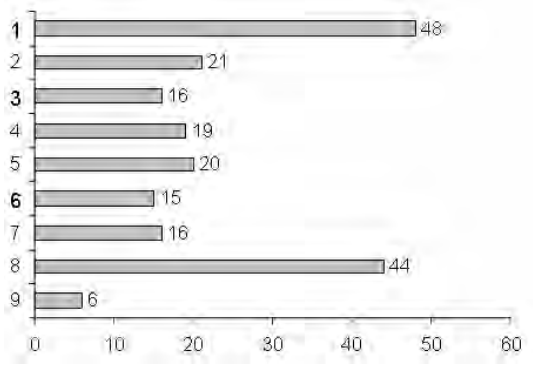


Fig. 6. Occurrences of errors grouped by their category

4 Conclusion and Future Work

The presented set of OCL constraints is the result of an intensive analysis of the semantics of MOF 2.0 as a whole. It is the result of a manual analysis since we are primarily interested in a semantical consistent version of MOF 2.0 for the application as graph schema language. We will use this set of constraints to complete the bootstrapping process of MOFLON. Therefore, we had to integrate a OCL 2.0 parser and code generator. After an evaluation of available and appropriate tools we decided to integrate the Dresden OCL toolkit [LO04]. Since the OCL code generator component is still work in progress we are only able to generate evaluation code for a simple subset of the constraints. After an adaptation of the constraints to the capabilities of the code generation we will generate an analysis component for the MOFLON framework which checks edited metamodels to their full compliance to MOF 2.0. The current integration of the toolkit is based on the provided integration interfaces. The long term aim is

to exchange the toolkit's model repository (MDR) with a integrated metamodel that is generated with MOFLON.

Acknowledgement

The authors are grateful to Sascha Rüter for his outstanding work that contributes to this report.

References

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

[BGG04] Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche. Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc. 7th Int. Conf. Unified Modeling Language (UML'2004)*, volume 3273 of *Lecture Notes in Computer Science (LNCS)*, pages 188–197, Berlin, 2004. Springer Verlag, Springer Verlag.

[Bic04] Bichler. *Codegeneratoren für MOF-basierte Modellierungssprachen*. PhD thesis, Universität der Bundeswehr München, 2004. German.

[Dir02] Ravi Dirckze. *Java^{TM} Metadata Interface (JMI) Specification, Version 1.0*. Unisys, June 2002.

[LO04] S. Löcher and S. Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. In P. Schmitt, editor, *Workshop Proc. OCL 2.0 - Industry standard or scientific playground?*, volume 102 of *Electronic Notes in Theoretical Computer Science*, pages 43–61. Elsevier, November 2004.

[Obj03] Object Management Group. *UML 2.0 Infrastructure Specification*, September 2003. ptc/03-09-15.

[Obj04] Object Management Group. *UML 2.0 Infrastructure Specification*, November 2004. ptc/04-10-14.

[Obj05a] Object Management Group. *OCL 2.0 Specification*, Juni 2005. ptc/2005-06-06.

[Obj05b] Object Management Group. *UML 2.0 Superstructure Specification*, August 2005. formal-05-07-04.

[Obj06] Object Management Group. *Meta Object Facility (MOF) Core Specification*, January 2006. formal/06-01-01.

[Rea05] Real-Time Systems Lab, Darmstadt University of Technology. *Set of OCL constraints for a consistent MOF 2.0*, 2005. http://www.es.tu-darmstadt.de/download/publications/amelunxen/Constraints.pdf.

[SDG+06] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A. Schürr, and A. Zündorf. *Das MATE Projekt - visuelle Spezifikation von MATLAB-Analysen und Transformationen*. Gesellschaft für Informatik, Bonn, 2006. submitted for publication, in German.

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.

Ambiguity issues in OCL postconditions

Jordi Cabot

Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya
Rbla. Poblenou, 156 E08018 Barcelona, Spain
jcabot@uoc.edu

Abstract: There are two different approaches to specify the behavior of the operations of an Information System. In the imperative approach, the operation effect is defined by means of specifying the set of actions (creation of objects and links, attribute updates…) to apply over the system state. With the declarative approach, the effect is defined by means of contracts stating the conditions that the system state must satisfy *before* (precondition) and *after* (postcondition) the operation execution.

From a specification point of view, the declarative approach is preferable. The main issue regarding declarative specifications is their ambiguity. Commonly, there are many different system states that satisfy an operation postcondition. However, in general, only one of them is the one the designer had in mind when defining the operation, and thus, that state should be the only one considered valid at the end of the operation execution. In this paper, we identify some of the common ambiguities appearing in OCL postconditions and provide a default interpretation for each of them in order to improve the usefulness of declarative specifications.

1. Introduction

A conceptual schema (CS) is the representation of the general knowledge of a domain. In conceptual modeling, we call Information Base (IB) the representation of the state of the CS (the set of existing objects and links) in the Information System.

The state of the IB changes due to the application of the modifying actions issued by the execution of the operations defined in the CS. An action is the fundamental unit of behavior specification [11]. Among the possible actions we have the creation of a new object or link, its deletion, the update of an attribute and so forth.

The effect of the operations can be specified in two different ways, *imperatively* or *declaratively* [16]. In an imperative specification the designer explicitly defines the set of actions to be applied over the IB. In a declarative specification the designer defines a contract for each operation. The contract consists of a set of pre and postconditions. A precondition defines a set of conditions over the operation input and the IB that must hold when the operation is invoked. The postcondition states the set of conditions that must be satisfied by the IB at the end of the operation execution. We assume that pre and postconditions are specified in OCL.

From a specification point of view, the declarative approach is preferable since it allows a more abstract definition of the operation effect and defers until a later stage most of the implementation issues [16]. In this sense, the imperative definition of an operation can be regarded as a lower-level definition of the operation effect. Moreover, declarative specifications are more concise than their imperative counterparts. This favours the readability of the contract specifications.

The main problem regarding declarative specifications is that they are *underspecifications* [16], i.e. in general there are several possible states of the IB that may verify the postcondition of an operation contract. This means that a declarative specification may have several equivalent implementations (i.e. imperative versions). We have a different version for each set of actions that, given a state of the IB verifying the precondition, evolve the IB to one of the possible states verifying the postcondition of the operation contract. This problem is not specific of OCL declarative specifications but common to other purely declarative languages used to specify operation contracts as JML, Eiffel or logic languages. However, it is especially relevant in OCL contracts because of the high expressiveness of the OCL.

The definition of a postcondition precise enough to characterize a single state of the IB is cumbersome and error-prone [5],[15]. For instance, it would require to specify in the postcondition all objects and links not modified by the operation (*frame problem* [5]). There are other ambiguities too. Consider a postcondition as $o.at_1=o.at_2$, where o represents an arbitrary object and at_1 and at_2 two attributes. Given an initial state of the IB, states obtained after assigning to at_1 the value of at_2 satisfy the postcondition. However, states where at_2 is changed to hold the value of at_1 or where the same value is assigned to both attributes satisfy the postcondition as well. Strictly speaking, all three interpretations are correct since all satisfy the postcondition. However, most probably, only the first one (where we assign the value of at_2 over at_1) represents the behaviour the designer meant when defining the postcondition.

We believe it is really important designers are aware of the different ambiguities included in a postcondition and of the whole set of possible states that satisfy the postcondition due to such ambiguities. Then, they may decide to extend/rephrase the postcondition in order to prevent some of the undesired states. This may imply to complement the postcondition with additional information regarding the parts of the system state that cannot be changed by the operation [4, 5], or even, to mix the declarative specification with imperative constructs [2].

In this paper we follow an alternative approach. For each ambiguous OCL expression that may appear in a postcondition we define its default interpretation. These default interpretations represent usual designers' assumptions about how that expression should be tackled when implementing/validating the operation. As an example, a default interpretation for the $X=Y$ ambiguous expression commented above is that X should take the value of Y. According to this interpretation, states where Y takes the value of X or where both take a different value, should be considered invalid.

The detection of common types of ambiguities and the proposal of a default interpretation for each of them are the main contributions of this paper. We believe that our approach offers several benefits. It improves the quality of the applications

by means of detecting possible errors in the specification of the operation contracts and by ensuring that the application behaviour is better aligned with the designer intentions. Moreover, it opens the possibility of leveraging current MDA and MDD methods and tools by allowing code-generation from declarative specifications (once translated to an equivalent imperative version) in the final technology platform. Until now, an automatic translation was not feasible because of the high number of possible imperative versions for each declarative specification. The default interpretation reduces the number of possible alternatives (to just one, in the best case) and could be used to guide the code-generation process. As an additional benefit, the discussion presented in the paper may help to achieve a deeper understanding of the semantics of operation contracts and their ambiguity problems.

The rest of the paper is structured as follows. Next section presents a set of ambiguous OCL expressions and provides their default interpretation. Section 3 covers some inherently ambiguous postconditions. Section 4 compares our approach with related work and, finally, section 5 present some conclusions and further research.

2. Ambiguities in operation contracts

Given the contract of an operation *op* and an initial state *s* of an IB (where *s* verifies the precondition of *op*) there exist, in general, a set of final states $set_{s'}$ that satisfy the postcondition of *op*. All implementations of *op* leading from *s* to a state $s' \in set_{s'}$ must be considered correct. Obviously, *s'* must also be consistent with the integrity constraints defined in the CS, but, assuming a strict interpretation of operation contracts [12], the verification of those constraints does need to be part of the contract.

Even though, strictly speaking, all states in $set_{s'}$ are correct, only a small subset $acc_{s'} \subset set_{s'}$ could probably be accepted as such by the designer. The other states satisfy the postcondition but do not represent the expected behaviour of the operation intended by the designer. In most cases $|acc_{s'}| = 1$ (i.e. from the designer's point of view there exists only a state *s'* that satisfies the postcondition).

The first aim of this section is to detect some common OCL expressions that, when appearing in a postcondition, increase the value of $|set_{s'}|$, that is, the expressions that introduce an ambiguity problem in the operation contract. We also consider the frame problem, which, in fact, appears because expressions required to specify conditions over the state of parts of the IB are missing in the postcondition.

Ideally, once the designer is aware of the ambiguities appearing in an operation *op*, he/she should define the postcondition of *op* precise enough to ensure that $acc_{s'} = set_{s'}$. However, this is not feasible in practice since then postconditions become much longer, cumbersome and error-prone [5],[15]. Therefore, the second aim of this section is to provide, for each ambiguity expression, a default interpretation that solves the ambiguity problem by selecting from $set_{s'}$ those that most probably represent the intention of the designer at the moment of writing that part of the postcondition.

The default interpretations express common assumptions used during the specification of operation contracts. They have been developed after analyzing many examples of operation contracts of different books, papers and case studies and comparing them, when available, with the operation textual description.

In what follows we present different ambiguity problems. We provide in the appendix a list of transformation rules we can apply over the original OCL expressions to extend the set of postconditions covered in this section.

2.1 Ambiguity 1: "State of objects and links not referenced in the postcondition"

In general, OCL expressions appearing in a postcondition only restrict the possible values of part of the elements of the IB (in particular, the ones referenced in the operation). The values of the rest of objects and links in the IB are left undefined, and thus, any state that modifies them is acceptable as long as the state verifies the postcondition.

Default interpretation: Nothing else changes.
This interpretation represents the most common adopted solution to the frame problem. It states that objects not explicitly referenced in the postcondition should remain unchanged in the IB. This implies that they cannot be created, updated or deleted during the transition to the new state of the IB. Similarly, links of associations not traversed during the evaluation of the postcondition cannot be created nor deleted.

Besides, for those objects that appear in the postcondition, only the properties (either attributes or association ends) mentioned in the postcondition definition may be updated.

2.2 Ambiguity 2: "Equality expressions"

By far, the most common operator in postconditions is the equality operator. Given an expression *X.a=Y.b* (where *X* and *Y* are two arbitrary OCL expressions and *a* and *b* two properties), there are three kinds of changes over the initial state resulting in a new state satisfying the expression. We can either assign the value of *b* to *a*, assign the value of *a* to *b* or assign to *a* and *b* an alternative value *c*.

Note that if either operand of the equality comparison is a constant value or is defined with the *@pre* operator then just a possible final state exists. Since the value of that operand cannot be modified, the only possible change is to assign its value to the other operand. This applies also to other ambiguities described in this section.

Default interpretation: The order of the operands in the equality expression reflects the desired change
We believe that the common interpretation for an expression *X.a=Y.b* is that *a* must take the value of *b*. This should be the only final state considered valid at the end of the operation. If a designer was meant to define that b should take the value of *a* he/she would have surely written the expression as *Y.b = X.a*. In the same way, if the desired final state was that the one where the value of *a* and *b* was equal to *c*, most

probably, he/she would have included in the postcondition the expression *X.a=c and X.b=c*.

2.3 Ambiguity 3: "if-then-else expressions"

An *if-then-else* expression evaluates to false when the *if* condition is satisfied but the *then* condition is evaluates to false or, reversely, when the *if* condition evaluates to false but the *else* expression is not satisfied. Therefore, given an *if-then-else* expression included in a postcondition *p*, there are two groups of final states that satisfy *p*: 1 – States where the *if* and the *then* condition are satisfied or 2 – states where the *if* condition evaluates to false and the *else* condition evaluates to true.

Default interpretation: Do not falsify the *if* clause.
We believe the desired behaviour for *if X then Y else* Z expressions is to evaluate *X* and enforce *Y* or *Z* depending on the value of X. According to this interpretation, states obtained by means of falsifying *X* do not represent the designer's intention. Implementations of postconditions that modify the *X* expression to ensure that *X* evaluates to false are not acceptable (even if, for some states of the IB, it could be easier falsifying *X* to always avoid enforcing *Y* instead of enforcing *Y* or *Z* depending on the value of *X*).

2.4 Ambiguity 4: "*includes and includesAll* expressions"

Given an initial state *s* and a postcondition of type *X->includesAll(Y)*, all final states where, at least, the objects of *Y* have been included in *X* satisfy the postcondition. However, states that, apart from the objects in *Y*, add other objects to *X* also satisfy the postcondition. Moreover, another possible group of final states that satisfy the expression are those where *Y* evaluates to an empty set, since by definition all sets include the empty set.

For *includes* expressions we follow the same reasoning. The only difference is that, for those expressions, *Y* does not return a set of objects but a single instance.

Default interpretation: Minimum number of insertions over the collection *X* and no changes over *Y*
Following this assumption, the new state *s'* should be obtained by means of adding to *s* the minimum number of links necessary to satisfy the operation postcondition. For expressions such as *X->includes(Y)* or *X->includesAll(Y)* (where *X* and *Y* are two arbitrary OCL expressions) a maximum of *X@pre->size()+Y->size()* links must be created. States including additional insertions are not acceptable.

States where *Y* is modified to ensure that it returns an empty result are neither acceptable.

This interpretation may seem similar to the one defined to deal with the frame problem. The difference is that there we addressed the creation and deletion of links of associations not referenced in the postcondition, while this one tackles minimal modifications over elements that are referenced in the postcondition.

2.5 Ambiguity 5: "*excludes and* excludes*All* expressions"

Given an initial state *s* and a postcondition of type *X->excludesAll(Y)*, all final states where, at least, the objects of *Y* have been removed from the collection of objects returned by *X* satisfy the postcondition. However, states that, apart from the objects in *Y*, remove other objects from *X* also satisfy the postcondition.

Additionally, states where *Y* does not return any object also satisfy the postcondition since then, clearly, *X* also excludes all the objects in *Y*.

For *excludes* expressions we follow the same reasoning. The only difference is that for those expressions *Y* does not return a set of objects but a single instance.

Default interpretation: Minimum number of deletions over the collection *X* and no changes over *Y*

According to this interpretation, the acceptable states are those where the new state *s'* is obtained by means of adding to the initial state *s* the minimum number of links necessary to satisfy the operation postcondition and where *Y* has not been modified to ensure that it returns an empty set.

For expressions like *X->excludes(Y)* or *X->excludesAll(Y)* a maximum of *X@pre->size()–Y->size()* may be deleted.

2.6 Ambiguity 6: "*forAll* iterators"

There are two possible approaches to ensure that an expression like *X ->forAll(Y)* (where *X* represents an arbitrary expression and *Y* a boolean expression) is satisfied in a new state of the IB. We can either ensure that, in the final state, all elements in *X* verify *Y* or to ensure that *X* results in an empty collection, since a *forAll* iterator over an empty collection always returns *true*.

Default interpretation: Do not empty the collection expression

The desired behaviour is to ensure that all elements in *X* verify the condition *Y* and not to force *X* to be empty.

2.7 Ambiguity 7: "*oclIsTypeOf* and *oclIsKindOf* operators"

The condition *obj.oclIsTypeOf(C)* requires type *C* to be one of the classifiers of *obj*. Therefore, new states where *C* is added to the list of classifiers of *obj* satisfy the condition, regardless of any other modifications to the list of classifiers of *obj*. States where additional classifiers have been added or some classifiers removed from *obj* also satisfy the condition.

Similarly with *obj.oclIsKindOf(C)* expressions. The only difference is that, for these expressions, we only require that *C* or one of its subtypes is added to *obj*.

On the contrary, conditions like *not obj.oclIsTypeOf(C)* establish that in the new state *obj* cannot be instance of *C* (and likewise with *not obj.oclIsKindOf(C)*, where *obj* cannot be instance of *C* or instance of one of its subtypes). Therefore all states

verifying this condition are valid even if they add/remove other classifiers from the list of classifiers of *obj*.

Default interpretation: Minimum number of specializations/generalizations
When defining this kind of expressions, we assume that the desired behaviour is just to express the minimum set of specializations/generalizations required to satisfy the postcondition. Therefore, new states where *obj* has been specialized also to other classifiers apart from *C* are not valid (unless required due to other expressions appearing in the postcondition). As an example, for expressions like *obj.oclIsKindOf(C)* only the classifier *C* or one of its subtypes may be added to *obj* during the transition to the new state.

For *not obj.oclIsTypeOf(C)* expressions, no other classifiers (apart from *C*) should be removed from *obj*. Similarly, conditions like *not obj.oclIsKindOf(C)* require that *obj* is generalized to a direct supertype of *C*. No other generalizations should be applied.

3. Inherently ambiguous postconditions

In some sense, all postconditions may be considered ambiguous since, in general, there are several states of the IB that verify a given postcondition. However, for most postconditions, the default interpretations presented in section 2 allow to disambiguate them by means of determining which state is the preferred among the possible ones.

Nevertheless, some postconditions are inherently ambiguous (also called non-deterministic [2]). We cannot define a default interpretation for them since, among all possible states satisfying the postcondition, there does not exists a state clearly more appropriate than the others. As an example assume a postcondition with an expression $a>b$. There is a whole family of states verifying the postcondition (all states where *a* is greater than *b*), all of them equally correct, even from the designer point of view or, otherwise, he/she would have expressed the relation between the values of *a* and *b* more precisely (for instance saying that $a=b+c$).

We believe it is worth to identify these inherent ambiguous postconditions since most times the designer does not define them on purpose but by mistake. Table 3.1 shows a list of expressions that cause a postcondition to become inherently ambiguous. The list is not exhaustive but contains the most representative ones.

Table 3.1 List of ambiguous expressions

Expression	Ambiguity description
post: B_1 ***or … or*** B_n	At least a B_i condition should be true but it is not defined which one/s
X<>Y, X>Y, X>=Y, ***X<Y, X<=Y***	The exact relation between the values of *X* and *Y* is not stated

X+Y=W+Z ***(likewise with -,*,/,…)***	The exact relation between the values of the different variables is not stated.
X->exists(Y)	An element of *X* must verify *Y* but it is not defined which one
X->any(Y)=Z	Any element of *X* verifying *Y* could be the one equal to *Z*
X->union(Y)=Z ***(likewise with ∩,−,…)***	There are at least 2^n different ways to distribute the elements of Z between X and Y (n=|Z|) to ensure that the expression is satisfied
X.p->sum()=Y	There exist many combinations of single values that once added may result in *Y*
X.n_1.n_2…n_n=Y	We can either assign *Y* to the object/s obtained at the end of the navigation n_n or to change an intermediate link to obtain at n_n an object/s equal to *Y*
$Op_1()$ = $op_2()$	The values returned by two operations must coincide. Depending on its body, there may be several alternative ways to satisfy this equality

4. Related Work

Ambiguity problems of declarative specifications have been poorly studied apart from the frame problem [5],[15] and a couple of basic assumptions regarding object (and collection) creations and removals [14]. The most usual strategy to deal with the ambiguity problems in declarative specifications forces the designer to explicitly state in the postconditions which elements of the IB change and which remain the same (this is the case of [5], [4] and formal languages as Z, VDM or Larch). More recent approaches, as [2], try to combine the OCL with imperative languages to permit designers specify more clearly the semantics of the contracts.

However, none of them tries to automatically disambiguate declarative specifications without burdening the designer with the task of defining additional information in the postconditions. Besides, as we have commented before, the specification of completely precise postconditions is not feasible in practice. Nevertheless, such approaches could be useful to deal with the problematic postconditions of section 3.

The support for declarative specifications in current CASE and MDA tool is rather limited. Most of them only deal with imperative specifications (see [10] as a representative example). There exist several OCL tools allowing the definition of operation contracts (see, among others, [3], [6], [8],[7]). However, during the code-generation phase, the contracts are simply added as validation conditions. Contracts are transformed into *if-then* clauses that check at the beginning and at the end of the operation if the pre and postconditions are satisfied (and raise an exception otherwise). An exception is [1] that is able to check the correctness of an implementation with respect to its contract. None of them considers the ambiguity problems of the declarative specification, any state satisfying the postcondition is

considered valid without considering that, in fact, the designer would regard some of these *valid* states as invalid ones.

5. Conclusions and further research

In this paper we have detected several OCL expressions that, when included in a postcondition specification, introduce ambiguity problems. We define that a postcondition is ambiguous when it is an underspecification, i.e. when there are several states of the IB that satisfy it, even though, most probably, the designer would only consider as valid states a (small) subset of them.

Therefore, regardless how the designer decides to handle these ambiguity issues, we believe it is important he/she be aware of them since they may even indicate an error in the declarative specification (especially for the inherent ambiguities of section 3). In the declarative specifications we analyzed and according to the contract definition in natural language, many times the designers were unaware of the ambiguities present in their postconditions.

Additionally, we propose an approach to automatically disambiguate the postconditions by means of providing a default interpretation for each kind of ambiguous expression. The default interpretation determines, from the possible states satisfying the ambiguous expression, the one/s that best represents the designer's intention when specifying the postcondition.

Our proposed interpretations require some strong assumptions about how the postconditions are specified, yet we believe the assumptions reflect the way designers tend to (unconsciously?) specify the postconditions. They have been validated against two case studies of real-life applications (a Car Rental System [9] and an e-marketplace system [13]) as well as with other examples appearing in different books, papers and tutorials. Nevertheless, we would like to have the opportunity to discuss them among the members of the OCL community in order to see whether they are accepted or alternatives ones should be proposed (or even if there is no agreement in the existence of such default semantics for postconditions). Obviously, our approach cannot be applied when the proposed assumptions are not followed since then we may end up restricting some possible final states that should be considered valid.

There are other directions in which we plan to continue our work. First, we plan to extend the set of ambiguous expressions we detect. To facilitate an empirical validation of our approach we are also interested in developing an animator tool that given an operation contract and a state of the IB, applies our default interpretations to the contract postcondition in order to compute the new state for the IB. Moreover, we want to explore the possibility of generating (semi) automatically the implementation of an operation starting from its declarative specification. This translation would be useful to leverage current MDA tools, which only support code-generation from imperative specifications.

Acknowledgments

Thanks to people of the GMC group (and especially to Anna Queralt) for their many useful comments to previous drafts of this paper. This work has been partly supported by the Ministerio de Educacion y Ciencia (project TIN 2005-06053) and by the Generalitat de Catalunya (grant 2006 BE 00062).

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P. H.: The KeY tool, Integrating object oriented design and formal verification. Software and Systems Modeling 4 (2005) 32-54
2. Baar, T.: OCL and Graph-Transformations - A Symbiotic Alliance to Alleviate the Frame Problem. In: Proc. MODELS'05 Workshop on Tool Support for OCL and Related Formalisms, Technical Report, LGL-Report-2005-001 (2005) 93-109
3. Babes-Bolyai University. Object Constraint Language Environment 2.0. http://lci.cs.ubbcluj.ro/ocle/
4. Beckert, B., Schmitt, P. H.: Program verification using change information. In: Proc. 1st Int. Conf. on Software Engineering and Formal Methods (SEFM'03), (2003) 91-101
5. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Transactions on Software Engineering 21 (1995) 785-798
6. Borland. Borland® Together® Architect 2006. http://www.borland.com/us/products/together/
7. Dresden. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net/index.html
8. Dzidek, W. J., Briand, L. C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Proc. MODELS 2005 Workshops, LNCS, 3844 (2005) 10-19
9. Frias, L., Queralt, A., Olivé, A.: EU-Rent Car Rentals Specification. LSI Research Report, LSI-03-59-R (2003)
10. Mellor, S. J., Balcer, M. J.: Executable UML. Object Technology Series. Addison-Wesley (2002)
11. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)
12. Queralt, A., Teniente, E.: On the Semantics of Operation Contracts in Conceptual Modeling. In: Proc. CAiSE Short Papers 2005, CEUR Workshop Proceedings, 161 (2005)
13. Queralt, A., Teniente, E.: A Platform Independent Model for the Electronic Marketplace Domain. LSI Technical Report, LSI-05-9-R (2005)
14. Sendall, S.: Specifying reactive system behavior. Phd. Thesis. Dir: A. Strohmeier. École Polytechnique Fédérale de Lausanne (2002)
15. Sendall, S., Strohmeier, A.: Using OCL and UML to Specify System Behavior. In: Object Modeling with the OCL, The Rationale behind the Object Constraint Language. Springer-Verlag (2002) 250--280
16. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys 30 (1998) 459-527

Appendix

This appendix provides a list of simple transformation rules between OCL expressions. These transformations help to extend the set of OCL expressions included in the ambiguity patterns of sections 2 and 3. We group the equivalences by the type of expressions they affect. The capital letters *X*, *Y* and *Z* represent arbitrary OCL expressions of the appropriate type. The letter *o* represents an arbitrary object.

Table A.1 List of substitution rules

Type	Rules	
Boolean types	X implies Y → if X then Y else true	(not X or Y) and (X or Z)→ if X then Y else Z
	A xor B → (A or B) and (not A or not B)	not (not A) → A
	not (A or B) → not A and not B	not (A and B) → not A or not B
	A or (B and C)→(A or B) and (A or C)	
Collection Types	X->count(o)>0 → X->includes(o)	X->count(o)=0 → X->excludes(o)
	Y->forAll(y1| X->count(y1)>0) → X->includesAll(Y)	Y->forAll(y1| X->count (y1)=0) → X->excludesAll(Y)
	X->size()=0 → X-> isEmpty()	X->size()>0 → X->notEmpty()
Predef. iterators	X->select(Y)->size()>0 → X->exists(Y)	not X->exists(Y) → X->forAll(not Y)
	X->reject(Y) → X->select(not Y)	X->one(Y) → X->select(Y)->size()=1
	X->select(Y)->size()=0 → X->forAll(not Y)	X->select(Y)->size()=X->size() → X->forAll(Y)
	X->select(Y)->forAll(Z) → X->forAll(Y implies Z)	X->select(Y)->exists(Z) → X->exists(Y and Z)
	not X->forAll(Y) → X->exists(not Y)	

UML/OCL – Detaching the Standard Library

D.H.Akehurst[1], W.G.J.Howells[1], K.D.McDonald-Maier[2]

[1] University of Kent
{D.H.Akehurst,W.G.J.Howells}@kent.ac.uk
[2] University of Essex
kdm@essex.ac.uk

Abstract. The Object Constraint Language (or variations of it) is increasingly being used as a text based navigation or expression language over Object-based modelling languages other than the original target of UML. The recent increase of Domain Specific Languages has in particular contributed to this process. As a consequence, it is useful to investigate the lengths to which an OCL like expression language can be made independent of the specifics of the underlying modelling language, concepts and implementation. This paper looks at the issue from the perspective of detaching the OCL specification from the standard library of types that is currently built into its definition.

1 Introduction

The Object Constraint Language (OCL) [3] was originally conceived and designed as a constraint language for use with the UML. Since then, its use has been extended to form a general expression and query language for use with MOF based modelling languages. Such languages need only to support a few basic concepts in order to facilitate the use of OCL for evaluating expressions over the language.

Based on the OCL standard, our works with the Kent OCL toolkit [2] have shown that it is possible to construct a "Bridge" to multiple modelling languages and / or multiple implementations of those languages. – UML, MOF, ECORE, MDR, Java..etc.

Our original bridge proposes a small set of interfaces that must be implemented in order to support the use of OCL with a new language. Some of these classes are required specifically to support the concepts in the OCL standard library, and thus the implementation is by necessity tied to that library. In addition, within the implementation of the OCL processor, the mapping from the OCL standard lib types to implementation types is fixed, e.g. an OCL *String* maps to a Java *String*.

When moving to different implementations of a model, one often discovers that the mapping of OCL standard library types onto similar types in the model implementation is different for different implementation techniques. For example, there may be a user defined String class that the OCL string should be mapped to. In our experience, we have found that it would be most useful to be able to replace the standard types (and methods available on those types) with alternatives, depending on the implementation of the model.

This would achieve three things:

1. The mapping to model implementation of basic types is simplified
2. It provides the '*user*' the option to extend, alter, or replace elements of the standard library.

3. It simplifies the implementation of an OCL compiler/interpreter.

The purpose of this paper is to examine the feasibility of defining an OCL-like language that would meet the requirement of being able to replace the standard library.

The paper is organised as follows: Section 2 looks at the basic requirements of a textual navigation language for object graphs. Section 3 discusses issues about mapping syntax and literal values onto a (potentially) user defined standard library. Section 4 presents a simple meta-model we have been experimenting with. Section 5 includes a discussion on the relationship between user models and the OCL-like language.

2 Basic requirements

Obviously we cannot, with OCL, provide a generic language for writing expressions over any language. However, if we make the not unreasonable assumption that the target languages will all be based on a notion of object-orientation, then we can assert that expressions in the language must be capable of navigating a path through a graph of objects and links. The following subsections introduce the main components we believe are necessary in such a language.

2.1 Navigation Paths

There are two basic requirements for providing a language for navigating such a graph.

1. A means to reference the stating point in a navigation path
2. A means to traverse a link.

A starting point is traditionally given by defining the 'type' of object that can be used as the starting point for the given expression. Further, there are traditionally two options for crossing a link to a new object, either via a property (attribute or association end in UML) or by an operation call. In some cases, the link is pre-defined as part of the initial graph (e.g. an association/link) or sometimes the link is dynamically constructed (e.g. as part of the execution of an operation or derived property). There is no relevant difference between qualified property and operation; the difference is really a syntactic one:

- object.property or object.property[qualifier]
- object.operation(argument)

From a navigation point of view, both of these result in a new 'node' (object) in the navigation path and could be seen as equivalent. However, with a closer look there are some differences that require us to treat them separately:

1. Due to various conventions, the implementation of operations and properties has tended to differ; e.g. the Java conventions of implementing properties with *accessor* and *mutator* methods, starting their name with '*get*' and '*set*'.
2. There is a semantic difference between an operation and a qualified derived property! An operation can modify the state of a model (i.e. modify the objects and links in the graph) whereas a derived property does not; from an OCL perspective, OCL is not supposed to alter the

state, and can thus only call operations that do not alter the state (e.g. marked with 'isQuery' in UML).

The second of these distinctions is not necessarily relevant if an extension of OCL is to be used as a textual means to describe behaviour, including state modification.

Our preference on these issues is to provide two mechanisms for navigation, operations and qualified properties (there may be no qualifier arguments); thus facilitating a differentiation between property and operation if necessary.

2.2 Iterator Expressions

One of the significant features of OCL is the "*iterator*" expressions. These are akin to higher order functions from functional programming languages. They are essential for the navigation over collections of objects. In general, these *iterator* operations take an argument, which is of the form of an OCL expression, and apply the expression to each element of the collection in order to calculate the result of the operation. In OCL, these *iterator* operations are currently built into the language, i.e. they are part of the language definition rather than operation defined on an object. This unfortunately means that users cannot define new *iterator* operations. In order to facilitate such definitions, it is necessary to introduce the concept of an *Expression* as an object type, so that it can be passed as an argument.

2.3 Alternative Paths

With a text expression, it is often necessary to define a set of alternative navigation paths, the choice of which to take being determined at runtime. Typically this facility is offered with concepts such as an "*if*" statement and/or a "*switch*" or "*select*" statement.

The OCL currently has the concept of an if statement; we see no reason not to extend this to the multiple path options offered by concepts such as the "switch" or "select" statements found in many programming languages. Such a concept could follow the same convention as the existing "*if*" statement in requiring a default option (the construct must always return a value), or assume that if a default option is not provided then an OCL "*null"* or "*invalid"'* value is returned.

This kind of multiple paths conditional statement is particularly useful when testing variables of an enumeration type, in addition to other situations, which using the current OCL must be formed using multiple nested "*if … then … else … endif*" statements.

2.4 Sub Expressions

The OCL concept of "*let … in …"* is essential to writing concise and readable complex expressions. They provide a means to define a number of sub-expressions that can be reused within the expression as a whole. This could be seen as syntactic sugar; however, its use can, in addition to improving readability, also improve the performance of evaluating the expression; hence we feel the concept should be included in the language definition.

3 Syntax and Literal Values

It is necessary to define a binding between literal values and types within the model. Usually this is built in to the language. However, we believe it is feasible to provide definitions/mappings at compile time.

An example mapping between literal values and types found in a Java implementation of a model could be given as shown in Table 1. This mapping would not, of course, provide the operations given by the standard OCL library. (The non-terminal names come from the grammar specification of the language.)

Literal non-terminal name	**Type**
stringLiteral	java.lang.String
integerLiteral	java.lang.Integer
realLiteral	java.lang.Double
booleanLiteral	java.lang.Boolean

Table 1

An alternative mapping shown in Table 2, could provide the standard OCL operations, but requires the model to be implemented in Java using the named types.

Literal non-terminal name	**Type**
stringLiteral	my.ocl.String
integerLiteral	my.ocl.Integer
realLiteral	my.ocl.Real
booleanLiteral	my.ocl.Boolean

Table 2

Operator symbols are also an issue; whether they are prefix, infix or postfix, they must to be mapped to appropriate operations on a type. This could be provided in a simple fashion by binding the operator symbols to operations names as illustrated in Table 3. The problem with this would be with operators such as '-', which has both a prefix and an infix meaning and hence would ideally be mapped to two different operation names – 'minus' and 'negate'.

Operator Symbol	**Operation Name**
+	plus
-	negate
-	minus
*	multiply
/	divide
and	and
or	or
%	modulus

Table 3

A more complex approach could be taken by binding the operator symbol and the operand types to operations on particular types, as indicated in Table 4. This approach

requires the mappings to explicitly reference the types on which the operators are applicable.

Types	Operator Symbol	Operation Name
{..Integer, ..Integer}	+	plus
{..Double, ..Double}	+	plus
{..String, ..String}	+	concatenate
{..Double, ..Double}	-	minus
{..Integer, ..Integer}	-	minus
{..Double }	-	negate
{..Integer }	-	negate
{..Boolean, ..Boolean }	and	and

Table 4

A third alternative would be to make a distinction in the mapping information between prefix, infix and postfix operators, but require the names of the implementing operation to be the same whatever type the operator is applied to.

Type	Operator Symbol	Operation Name
infix	+	plus
infix	-	minus
prefix	-	negate
prefix	not	not
infix	and	and

Table 5

The first option is nice and simple, but is too restrictive; the second option gives us extensive flexibility in mapping operators to operations, but in out opinion requires too much information to be specified. Our preference is for the third option, which gives sufficient flexibility without requiring the level of detail necessary with the second option.

4 A Simple Navigation Language Meta-Model

The meta-model for a language defined along these means could consist of two parts

1. Navigation Part: for defining navigation paths and expressions.
2. Bridge Part: for abstracting the connection between navigation paths and the object graph.

The meta-model of an OCL-like language that we have been using to experiment with the ideas discussed in this paper is shown below in the two figures.

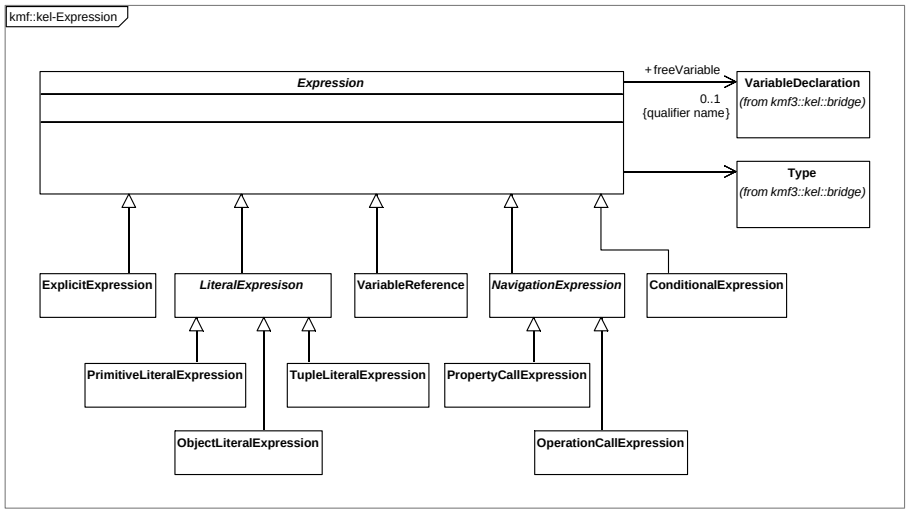


Figure 1 – Types of Expression

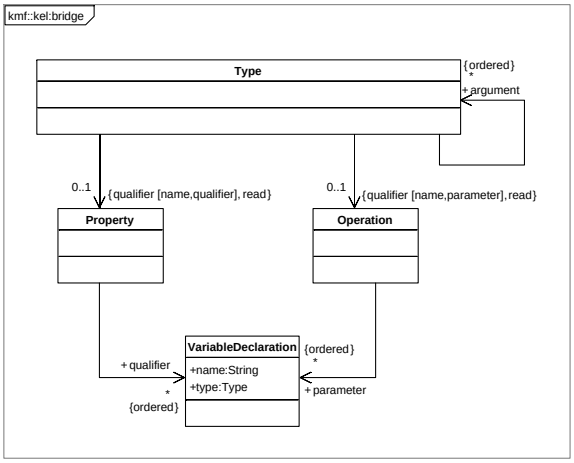


Figure 2 – Bridge Concepts

5 On Models

The OCL standard does not only provide a library of primitive and collection types and define a text based navigation language. It also alters the type hierarchy of the target execution model. That is to say, the OCL language makes an assertion that all types in the model extend types in the OCL standard library – e.g. OclAny and OclModelElement.

These types contain useful operations such as testing for equality or checking the type of an object. The assumption in OCL is that these operations are not provided by the model, and thus a common super type, that does provide these operations, is

required. This probably arose initially due to OCL being “added” to UML as an afterthought. We would argue that it is an unusual policy.

A more usual approach is to provide a common super type and ensure that every model type does extend it. For instance, common OO programming languages such as Java [4] and C# [1] have common super types of ‘java.lang.Object’ and ‘’System.Object’; all classes defined in these languages automatically extend the common super type, it is not added as an afterthought by the expression part of the language.

We propose that the same approach should be taken in the modelling world. All models must specify the type in the model that is the ‘common super type’ of the model; this type could be provided by a standard library, but may be replaced by an alternative.

The OCL standard library thus becomes a ‘model’ just like any other. However, this model will probably be included (imported) by specific domain models, providing a standard set of primitive and collection types and a standard common super type.

5.1 UML + OCL

The current situation of UML + OCL can be mimicked using the techniques discussed above as follows:

- The current OCL standard library is provided as a UML package and classes.
- A specific Domain Model includes the OCL standard library and defines the OclModelElement class as the root type for the model.
- The ‘new’ standard library will contain a type ‘*Expression’*.
- Iterator operations are defined on the collection classes, taking parameters of type *Expression*.

5.2 A New Standard Library

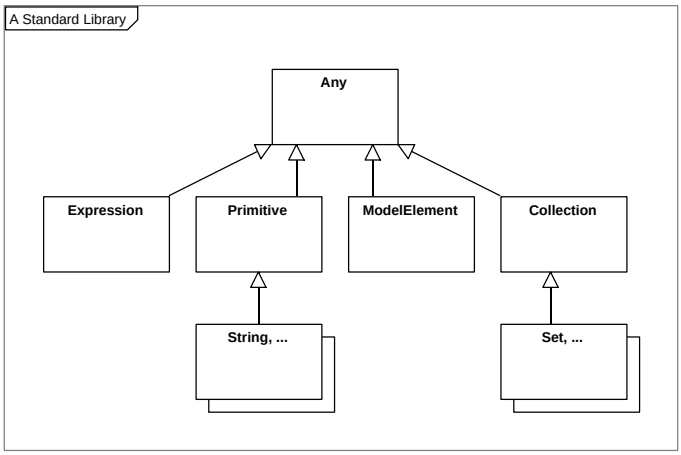


Figure 3 – A possible Standard Library

- **Expression**: An expression type enables us to pass expressions as parameters to operations.
- **Primitive**: A Primitive type is distinguished from other types as primitive objects can be constructed from string constants (e.g. by the OCL compiler).
- **ModelElement**: Within UML, a Model Element object embodies some notion of containment (i.e. composite/part structure indicated with black diamonds) and its type is given as a class in the model. This provides a candidate for the "*inheritance root*" of Models.
- **Collection**: A collection object does not require a notion of containment.

6 Conclusion

We have proposed an approach to "detaching" the OCL standard library from the navigation and expression part of the language. This moves the standard library (and types) to the same position as all other model elements, thus facilitating the extension or replacement of the standard types.

Our initial experiments based on the Kent OCL library have shown that this approach to providing OCL support has some merit; in particular we have found it very useful to be able to extend and replace the standard types.

An outcome of these experiments has been the production of a Java library that supports all the OCL iterator operations as operations on a collection classes; we have found that this library is very useful as a target for OCL code generation tasks, in addition to simply being a useful library for use within straightforward Java programmes.

We are currently experimenting with "bridging" the OCL-based navigation language to alternative object-based DSLs, alternative implementations of UML/MOF and with alternative libraries of primitive and collection types.

References

1. Microsoft: Visual C#. http://msdn.microsoft.com/vcsharp/
2. OCL-team: Kent OCL library. www.cs.kent.ac.uk/projects/ocl
3. OMG: UML 2.0 OCL Specification version 2.0. Object Management Group, pct/05-06-06 (June 2005)
4. Sun: Java Technology. http://java.sun.com/

Semantic Issues of OCL: Past, Present, and Future

Achim D. Brucker, Jürgen Doser and Burkhart Wolff

Information Security, ETH Zurich, 8092 Zurich, Switzerland
{brucker,doserj,bwolff}@inf.ethz.ch

Abstract We report on the results of a long-term project to formalize the semantics of OCL 2.0 in Higher-order Logic (HOL). The ultimate goal of the project is to provide a formalized, machine-checked semantic basis for a theorem proving environment for OCL (as an example for an object-oriented specification formalism) which is as faithful as possible to the original informal semantics. We report on various (minor) inconsistencies of the OCL semantics, discuss the more recent attempt to align the OCL semantics with UML 2.0 and suggest several extensions which make, in our view, OCL semantics more fit for future extensions towards program verifications and specification refinement, which are, in our view, necessary to make OCL more fit for future extensions.

1 Introduction

In research communities, UML/OCL has attracted interest for various reasons:

1. it is a formalism with a "programming language face," which is perhaps easier to adopt by software developers notoriously hostile to mathematical notation,
2. it puts forward the idea of an object-oriented specification formalism, turning objects and inheritance into the center of the modeling technique, and
3. it provides in many respects a "core language" for object-oriented modeling which makes it a good target for research of object-oriented semantics.

Item 1 refers not only to syntax, but also to semantics: OCL semantics comprises the notion of undefinedness to model exceptional computations abstractly; this is deeply integrated into the logics and presents a particular challenge to deductive systems. Further, especially item 2 makes OCL rather different from logical languages such as first-order logics (FOL), higher-order logics (HOL), set theory and derived specification formalisms such as Z [29,3] or VDM, which, following a long platonic tradition in logics, start with the notion of *values* and then model (hierarchies of) relations over them. On the other hand, this remarkably different perspective makes OCL semantics (and object-oriented specification as a whole) difficult; numerous luke-warm attempts to integrate object-orientation into specification formalisms, such as VDM++ or Object-Z, report—among many useful things—on this particular difficulty. Comparing OCL with the two related approaches JML and Spec#, the main difference is that OCL attempts to abstract

from concrete object-oriented programming languages, while JML and Spec# are designed as annotation-languages for them. This also holds for the UML Action package, which provides a deliberately abstract programming notation for "methods" associated to operations in class diagrams.

These three essentials motivated a long-term project to formalize the semantics of OCL 2.0 using HOL, leading to the proof environment HOL-OCL built on top of Isabelle/HOL [1,6]. The ultimate goal of the project is to provide calculi and automated proof support for reasoning over OCL formulae based on rules derived from this formalized semantics. This paves the way for proving the consistency of specifications, the proof-obligations resulting from specification refinements as well as the correctness of the transition to executable code. In this paper, we will present a by-product of this line of research: namely various formalization problems that we found or that we foresee when heading for an integrated verification method ranging from specifications to programming code. Extending earlier work [4], we report on a substantially larger range of problems and put it into perspective to recent developments of the OCL semantics.

2 Methodology: "Strong" Formal Semantics

In this section, we describe the foundations, the relevant techniques and the benefits of the methodology underlying HOL-OCL. This methodology boils down to provide a "strong," i.e., machine-checked and conservative, formalization of the standard's "Semantics" chapter [24, Appendix A]. The question may arise why this original formalization is not adequate for our goals. There are two reasons:

The fundamental reason results from the fact that [24, Appendix A] is based on naive set theory and an informal notion of "model." It assumes a universe for values and objects and algebras over it without any concern of existence and consistency. This paper-and-pencil semantics cannot be strongly formalized in this form, neither in an untyped set theory like Isabelle/ZF or a typed set theory residing in Isabelle/HOL. Since OCL is a typed language at the end, and since we wanted to have type-issues handled by the Isabelle type-checker and not inside the logic representation, it seems more natural to opt for a typed meta-language (like HOL).

The technical reason is a consequence of our design choice to represent the types of OCL expressions one-to-one by HOL types (i.e., the map is injective) such that only well-typed OCL formulae exist in the semantic representation in HOL. Consequently, all well-formedness-related side-conditions are unnecessary in calculi. Together with the fact that the Isabelle/HOL library can be re-used to a certain extent, this greatly improves the practicability of our approach. Technically speaking, our representation is a so-called *shallow embedding* without an explicit datatype for syntax and an explicit semantic interpretation function I mapping syntactic terms to a semantic domain.

In the following, we present our meta-language HOL and the underlying conservative methodology in more detail. We outline the shallow representation and show its equivalence to [24, Appendix A].

2.1 Higher-order Logic

Higher-order Logic (HOL) [8,2] is a classical logic with equality enriched by total parametrically polymorphic higher-order functions. It is more expressive than first-order logic, e.g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers.

HOL is based on the typed λ-calculus—i.e., the *terms* of HOL are λ-expressions. Types of terms may be built from *type variables* (like α, β, ...), optionally annotated by Haskell-like *type classes* as in $\alpha :: order$ or $\alpha ::$ bot) or *type constructors* (like bool or nat). Type constructors may have arguments (as in α list or α set). The type constructor for the function space $\Rightarrow$ is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ have the alternative syntax $[\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow$ bool, where bool is the fundamental logical type. We use infix notation: instead of $(\_ = \_)\ E_1\ E_2$ we write $E_1 = E_2$. The logical connectives $\_ \wedge \_$, $\_ \vee \_$, $\_ \rightarrow \_$ of HOL have type $[\text{bool}, \text{bool}] \Rightarrow$ bool, $\neg \_$ has type bool $\Rightarrow$ bool. The quantifiers $\forall \_.\_$ and $\exists \_.\_$ have type $[\alpha \Rightarrow \text{bool}] \Rightarrow$ bool. The quantifiers may range over types of higher order, i.e., functions or sets.

The type discipline rules out paradoxes such as Russel's paradox in untyped set theory. Sets of type α set can be defined isomorphic to functions of type $\alpha \Rightarrow$ bool; the definition of the elementhood $\_ \in \_$, the set comprehension $\{\_.\_\}$, $\_ \cup \_$ and $\_ \cap \_$ is then standard.

The *modules* of larger logical systems built on top of HOL are Isabelle *theories*. Among many other constructs, they contain type and constant declarations as well as axioms. Since stating arbitrary axioms in a theory is extremely error-prone and should be avoided, only very limited forms of axioms should be admitted and the side-conditions (both syntactical and semantical) checked by machine. These fixed blocks of declarations and axioms described by a syntactic scheme are called *conservative theory extensions* since any extended theory is consistent ("has models") provided the original theory was. Most prominent in the literature are *constant definition* and *type definition*. For example, a constant definition consists of a declaration declaring constant c of type τ and the (well-typed) axiom of the form: $c = E$ with the side-condition that c has not been previously declared, E does neither contain free variables nor c (no recursion). A further side-condition forbids type variables in the types of constants in E that do not occur in the type τ. As a whole, a constant definition can be seen as an "abbreviation," which makes the conservativity of the construction plausible (see [10] for details). The idea of an "abbreviation" is also applied to the conservative *type definition* of a type $(\alpha_1, \ldots, \alpha_n)T$ from a set $\{x \mid P(x)\}$.

The entire Isabelle/HOL library, including typed set theory, well-founded recursion theory, number theory and theories for data-structures like pairs, type

sums and lists is built on top of the HOL core-language by conservative definitions and derived rules. This methodology is also applied to HOL-OCL.

2.2 Formal Semantics Preliminaries in HOL

In OCL, the notion of explicit undefinedness plays a fundamental role, both for the logical and non-logical expressions:

> Some expressions will, when evaluated, have an undefined value. For instance, typecasting with `oclAsType()` to a type that the object does not support or getting the `->first()` element of an empty collection will result in undefined. *(OCL Specification [24], page 15)*

Thus, concepts like *definedness* and *strictness* play a major role in the OCL. We use a *type class* bot to specify the class of all types that contain the undefinedness element $\bot$. For all types in this class, we define a combinator strictify by:

$$\text{strictify}\, f\, x \equiv \text{if}\, x = \bot\, \text{then}\, \bot\, \text{else}\, f\, x$$

with type $(\alpha :: \text{bot} \Rightarrow \beta :: \text{bot}) \Rightarrow \alpha \Rightarrow \beta$. The operator strictify yields a strict version of an arbitrary function f.

Further, we use the type constructor $\tau_\bot$ that assigns to each type τ a type *lifted* by $\bot$. Per construction, each type $\tau_\bot$ is in fact in the type class bot. The function $\lfloor\_\rfloor : \alpha \to \alpha_\bot$ denotes the injection, the function $\lceil\_\rceil : \alpha_\bot \to \alpha$ its inverse for defined values.

On the expression level, lifting combinators defining the distribution of *contexts* or *environments* (see below) are defined as follows:

$$\begin{array}{ll}
\text{lift}_0\, f \equiv \lambda\, \tau.\ f & \text{of type } \alpha \Rightarrow V_\tau(\alpha)\,, \\
\text{lift}_1\, f \equiv \lambda\, X\ \tau.\ f(X\ \tau) & \text{of type } (\alpha \Rightarrow \beta) \Rightarrow V_\tau(\alpha) \Rightarrow V_\tau(\beta)\,, \text{ and} \\
\text{lift}_2\, f \equiv \lambda\, X\ Y\ \tau.\ f(X\ \tau)(Y\ \tau) & \text{of type } ([\alpha, \beta] \Rightarrow \gamma) \Rightarrow [V_\tau(\alpha), V_\tau(\beta)] \Rightarrow V_\tau(\gamma)\,.
\end{array}$$

where $V_\tau(\alpha)$ is a synonym for $\tau \Rightarrow \alpha$. The types of these combinators reflect their purpose: they "lift" operations from HOL to semantic functions that are operations on contexts.

2.3 Textbook vs. Combinator Style Semantics of Operations

In HOL-OCL, we use a combinator-style presentation of the semantic functions rather than a textbook-style presentation as used in the OCL standard, both for reasons of conciseness as well as accessibility to advanced techniques of automatic generation of library theorems [5]. In combinator style as used in the HOL-OCL libraries, for example, the constant 1, the unary operation `not _`, and the binary operation $\_ + \_$ are represented by the following constant definitions:

$$\begin{array}{rl}
1 \equiv & \text{lift}_0(\lfloor 1 \rfloor) \\
\texttt{not}\ _ \equiv & \text{lift}_1(\text{strictify}(\lfloor_\rfloor \circ (\neg_) \circ \lceil_\rceil) \\
_ + _ \equiv & \text{lift}_2(\text{strictify}(\lambda\, x.\ \text{strictify}(\lambda\, y.\ \lfloor \lceil x \rceil + \lceil y \rceil \rfloor)))
\end{array}$$

where $\_ \circ \_$ denotes function composition. We use overloading here: the $\_ + \_$ on the left-hand side of the last definition has type $[V_\tau(\text{int}_\bot), V_\tau(\text{int}_\bot)] \Rightarrow V_\tau(\text{int}_\bot)$ (where $V_\tau(\text{int}_\bot)$ is the HOL equivalent to the OCL type `Integer`), while the $\_ + \_$ on the right-hand-side has type $[\text{int}, \text{int}] \Rightarrow \text{int}$. This definition directly translates the idea that $\_ + \_$ in HOL-OCL is the strictified version of the "mathematical" $\_ + \_$ lifted over contexts.

The question arises why this definition is equivalent to the formalized version of the semantics given in the standard. The OCL 2.0 standard presents a definition scheme for all *strict basic operations* by just one example. For the +-operator on integers, [24, page A-11] presents this definition as:

$$I(+)(i_1, i_2) = \begin{cases} i_1 + i_2 & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot\,, \\ \bot & \text{otherwise.} \end{cases}$$

This semantic function for basic operations is integrated in the more general semantic interpretation function for OCL expressions like

> Let Env be the set of environments $\tau = (\sigma, \beta)$. The semantics of an expression $e \in \text{Expr}_t$ is a function $I[\![e]\!] : \text{Env} \to I(t)$ that is defined as follows.
> iv. $I[\![w(e_1, \ldots e_n)]\!]\tau = I(w)(\tau)\big(I[\![e_1]\!](\tau), \ldots, I[\![e_n]\!](\tau)\big)$
> *(OCL Specification [24], page A-26, definition A.30)*

Here, τ refers to the environment (in the sense of the standard), i.e., a pair consisting of a map β assigning variable symbols to values and a pair σ of system states.

There are two more semantic interpretation functions; one concerned with path expressions (i.e., *attribute and navigation expressions* [24, Definitions A.21], and one concerning the interpretation of pre and postconditions $\tau \models P$ which is used in two different variants.

To show the equivalence of the two formalization styles, we re-introduce a kind of "explicit semantic function" $I[\![E]\!]\tau$ into our shallow embedding as a syntactic marker, i.e., by stating the identity:

$$I[\![x]\!] \equiv x \qquad \text{with type } \alpha \Rightarrow \alpha\,.$$

For the addition over `Integer`, we prove the following theorem that explicitly states that our defined operator is an instance of the informal definition scheme in the standard:

$$I[\![X + Y]\!]\tau = \begin{cases} \lfloor \ulcorner I[\![X]\!]\tau \urcorner + \ulcorner I[\![Y]\!]\tau \urcorner \rfloor & \text{if } I[\![X]\!]\tau \neq \bot \text{ and } I[\![Y]\!]\tau \neq \bot, \\ \bot & \text{otherwise}\,. \end{cases}$$

The proof in HOL-OCL is simple and canonical: it consists of the unfolding of all combinator definitions and the syntactic marker I. The combinators are just abbreviations of re-occurring patterns in the textbook style definitions.

In the following, we summarize the differences between the OCL standards textbook definitions and our combinator-style approach:

1. The standard [24, chapter A] assumes an "untyped set of values and objects" as semantic universe of discourse. Since we reuse the types from the HOL-library to give Booleans, Integers and Reals a semantics, meta-expressions like $\{\text{true}, \text{false}\} \cup \{\bot\}$ used in the standard are simply illegal in our interpretation. This makes the injections $\lfloor\_\rfloor$ and projections $\lceil\_\rceil$ necessary.
2. The semantic functions in the standard are split into $I(x)$, $I[\![e]\!]\tau$, $I_{\text{ATT}}[\![e]\!]\tau$ and $\tau \vDash P$. Since we aim at a shallow embedding (which ultimately suppresses the semantic interpretation function), we prefer to fuse all these semantic functions into one.
3. The *environment* τ in the sense of the standard is a pair of a variable map and a pair of pre and post state. The variable map is superfluous in a shallow embedding (binding is treated by using higher-order abstract syntax), our contexts τ just consist of the state pair.

Of course, this presentation here covers only one aspect of the compliance of the HOL-OCL semantics to the standard for a tiny portion of the language; for an in-depth discussion for the complete language, the reader is referred to [6].

2.4 The Benefits of a "Strong" Formal Semantics

Our strong formalization of [24, Appendix A] has the following benefits:

A Consistency Guarantee. Since all definitions in our formal semantics are conservative and all rules are derived, the consistency of HOL-OCL is reduced to the consistency of HOL for the *entire language*.

A Technical Basis for a Proof-Environment. Based on the derived rules, control programs (i.e., *tactics*) implement automated reasoning over OCL formulae; together with a compiler for class diagrams, this results in a general proof environment called HOL-OCL. Its correctness is reduced to the correctness a (well-known) HOL theorem proving system.

Proofs for Requirement Compliance. The OCL standard contains a collection of formal requirements in its mandatory part with no established link to the informative part [24, Appendix A]. We provide formal proofs for the compliance of our OCL semantics with these requirements (see [6] for details).

Formalization Experience. Since our semantics is machine-checked, we can easily change definitions and check properties of them allowing for increased knowledge of the language as a whole.

3 The Past

OMG standards are developed in an open process by the OMG (Object Management Group) leading to a variety of (intermediate) "standardization" documents. Especially for UML and OCL, which have a long history. OCL was introduced as an OMG specification language as additional document [22] completing the UML 1.1 standard [23]. In later releases of the UML standards of the version 1.x series the OCL standard was a chapter of the UML specification, e.g., [25, Chapt. 6].

All the different versions of OCL 1.x are very close to each other, containing mainly an informal motivation of the indented use and semantics[1] of OCL together with a formal grammar of its concrete syntax. Understandably, these past version of the standard lacked many desirable features, e.g., the use of OCL was mainly limited to annotate class diagrams, no abstract syntax was included. Moreover, reading the OCL 1.x standards leaves more questions open than it answers. These shortcomings and open questions, like the handling of undefinedness, or recursion, were discussed [28,21,14,9] in academia and this discussions clearly fertilized the development towards OCL 2.0. Especially the work of Richters [27] in developing a formal semantics served as formal underpinning of the OCL 2.0 development. It was a major break-through in the process of defining a formal semantics for OCL . Many problems, like the handling of undefinedness, were clarified during the OCL 2.0 standardization process, some questions however, like the handling of recursion, are still unsolved.

4 The Present

4.1 The OCL 2.0 Standard

In this section, we give a brief overview of the chapters of the standard that are related with the semantics of OCL 2.0: first, the OCL standard is divided into *normative* parts and *informative*, i.e., not normative, parts. The semantics of the standard appears in the following chapters of [24]:

Chapter 7 "OCL Language Description": This *informative* chapter motivates the use of OCL and introduces it informally, mostly by examples.

Chapter 10 "Semantics Described using UML": This *normative* chapter describes the "semantics" of OCL using the UML itself. Merely an underspecified "evaluation" environment is presented.

Chapter 11 "The OCL Standard Library": This *normative* chapter is, in our opinion, the best source of the normative part of the standard describing the intended semantics of OCL. It describes the semantics of the OCL expressions as requirements they must fulfill.

Appendix A "Semantics": This *informative* appendix, based on [27] , defines the syntax and semantics of OCL formally in a textbook style paper-and-pencil notion.

We see the semantic foundations of the standard critical for several reasons:

1. The normative part of the standard does not contain a formal semantics of the language.
2. The consistency and completeness of the formal semantics given in "Appendix A" is not checked formally.
3. There is no proof, neither formal nor informal, that the formal semantics given in the informative "Appendix A" satisfies the requirements given in the normative chapter 10.

[1] A good overview of the different usages of the word "semantics" is given in [15].

Nevertheless, we think the OCL standard [24] ("`ptc/03-10-14`") is mature enough to serve as a basis for a machine-checked semantics and formal tools support. More recent versions, especially ("`ptc/06-05-01`"), are an ad-hoc attempt to align OCL 2.0 to the UML 2.0 and represent a considerable step back with respect to consistency and potential for formal semantics. Nevertheless, all issues addressed in this paper are also valid for "`ptc/06-05-01`".

In the remainder of this section, we will explain some selected problems; our choice focuses on semantical problems which, among others, are caused by inconsistencies or missing concepts in the standard document.

4.2 Implies

Recall that the OCL logic is based on a strong *Kleene Logic*. Consequently, most operators of the logical type like `_and_` are explicitly stated exceptions from the "operations are strict"-principle. In this section, we will discuss the `implies` operation in more detail. Its semantic is defined in the standard as follows:

1. [24, Chapter 11] requires the following specification of implies:

```
context Boolean::implies(b:Boolean):Boolean
  post: (not self) or (self and b)
```

2. [24, Appendix A] defines the `b1 implies b2` by a truth table:

b1 ╲ b2	false	true	⊥
false	true	true	true
true	false	true	⊥
⊥	⊥	true‡	⊥

While we were checking the consistency of the formal semantics [24, Chapter A] with the normative requirements [24, Chapter 11], we detected an inconsistency: calculating the truth table for the definitions of `implies` given in the normative part one would expect ⊥ instead of `true` on the position marked with an ‡. This inconsistency could be changed either by changing the truth tables [24, Chapter A] or by changing the requirements [24, Chapter 11] to:

```
context Boolean::implies(b:Boolean):Boolean
  post: (not self) or b
```

which represents the "classical definition" of implication.

Whereas different variants for implications for three-valued logics are considered in the literature [13,16], an analysis of the consequences for proof calculi reveals some bad surprises. For example, consider the usual assumption rearrangement rules valid in the "classical definition":

$$((X \texttt{ or } Y) \texttt{ implies } Z) = (X \texttt{ implies } Z) \texttt{ and } (Y \texttt{ implies } Z)$$
$$((X \texttt{ and } Y) \texttt{ implies } Z) = (X \texttt{ implies } (Y \texttt{ implies } Z))$$
$$X \texttt{ implies } (Y \texttt{ implies } Z) = Y \texttt{ implies } (X \texttt{ implies } Z)$$

which *do not hold* for the standard's definition of the implication. Although the choice made in the normative part of the standards is feasible, in the light of

these dramatic algebraic deficiencies, we qualify it as glitch from the deduction point of view and suggest to apply the definition used in the appendix.

4.3 Smashed Datatypes

The OCL standard defines all operations as strict, i.e., the evaluation of an operation is undefined if one of its argument is undefined. Nevertheless, there are two important exceptions to this rule: the logical connectives and the collection constructors. Whereas for the logical connectives this exception is stated both in the normative part [24, Chapter 11] and in the informative part [24, Appendix A], for the collection constructors this is only explained in the informative part [24, Appendix A]. The normative part of the standard does not cover this issue.

In the literature, sets with strict constructors are called *smashed*. Such smashed set types often occur in semantics for programming languages, e.g., SML. In a language with semantic domains providing $\bot$-elements, the question arises how they are treated in type constructors like product, sum, list or sets. Two extremes are known in the literature; for products, for example, we can have:

$$(\bot, X) \neq \bot \qquad \{a, \bot, b\} \neq \bot \qquad \ldots$$

or:

$$(\bot, X) = \bot \qquad \{a, \bot, b\} = \bot \qquad \ldots$$

The latter variant is called *smashed product* and *smashed set*. The normative chapters make no clear decision here. We strongly opt for a smashed collection semantics, based on two reasons:

1. OCL tends to define its constructs towards executablility and proximity to object-oriented programming languages such as Java, and more important
2. OCL with non-smashed collection semantics leads to very complicated logical calculi. Just consider the rule

$$\frac{\texttt{self.OclIsDefined()}}{\texttt{self->forAll(e | e.OclIsDefined())}}$$

   which only holds for a smashed semantics. Without such rules, reasoning over navigations, i.e., collections, always requires a proof of the definedness of all elements of a navigation.

To study the effects of a non-smashed collection semantics on formal reasoning, we provide a separate configuration of HOL-OCL, details can be found in [6].

4.4 Overloading and Late Binding

The concept of method-overloading is not yet fully supported by OCL. We believe, this is more or less due to some accidental circumstances:

1. The UML standard [25, chapter 4.4.1] requires that operation names are unique within the same namespace. In particular, subclasses may not overwrite inherited operations. Albeit, the UML standard allows one to (explicitly) overwrite *methods*, i.e., *implementations* of operations.
2. The OCL standard [24, chapter 7.3.41] restricts the use of the precondition and postcondition declarations to operations or other behavioral features. Sadly, all OCL tools we know of do not support the specification of preconditions and postconditions for methods.
3. While the OCL standard speaks in several places of operation calls, it does not give an hints how operation overloading should be resolved, neither does it explain in detail concepts like operation (method) calls or operation (method) invocations.

Bringing these items together, one has to conclude that operation overloading, and thus late-binding, is underspecified, or even not supported in OCL. Nevertheless, we think that overwriting inherited operations or methods is a very important feature of object-orientation and should be supported by the OCL: since operation calls can occur in OCL constraints, their meaning depends on the semantics of operation invocation. Thus we provide the theoretical foundations for supporting late-binding (and thus overloading of operations) within HOL-OCL [6], nevertheless a concrete syntax for specifying this has to be worked out. As simple workarounds, one can ignore the well-formedness constraint of UML for operations that requires operation names to be unique within one namespace. This is what most case-tools do.

Since the UML definition expresses in several places a clear preference for overloading operations, we suggest to extend the current OCL standard by a late-binding semantics of method invocation. We are aware that checks for conservativity will impose restrictions on invocations here to be discussed in subsection 4.6.

4.5 Equalities

Historically, object-oriented systems are equipped with a variety of different "equalities" [18]. Answering the question whether two objects are equal is not so obvious. For example, are two objects equal only if their object identifiers are equal (are they the same object?!) or are two objects equal if their values are equal? Whereas in traditional specification formalisms the equality is defined over values, the most basic equality over objects is the reference equality or identity equality, which is also the kind of equality that is usually provided as a default, i.e., "built-in," equality in object-oriented programming languages. Thus there is a fundamental difference between values and objects.

This situation, i.e., which role do references play within OCL, is not clearly stated in the OCL standard. ([6] gives a detailed discussion of this topic) and we will only discuss in this paper the consequences of taking undefinedness, e.g., values and references can be undefined, into account. Further, the well-known equivalence properties need to be generalized, e.g., *symmetry* ($x = y \Rightarrow y = x$) is generalized to *quasi-symmetry* ($x = y \Rightarrow y = x$ for x and y being defined).

Naturally, we can apply the concept of strictness to an equality operator: an equality operator is called *strict equality* if it evaluates to undefined whenever one of its arguments is undefined, i.e., if the following properties hold:

$$(o \doteq \bot) = \bot\,, \qquad (\bot \doteq o) = \bot\,, \text{ and } \qquad (\bot \doteq \bot) = \bot\,.$$

In contrast, an equality operator is called a *strong equality* if it satisfies the property: $(\bot \triangleq \bot) = \mathsf{T}$.

The OCL standard defines equality as the strict equality over values [24, Sec. A.2.2], and since objects are values, and object identifiers are not distinguished from object values [24, Definition A.10] we chose the strict equality $\_ \doteq \_$ as the default OCL equality within HOL-OCL. Nevertheless, several interesting properties, like being quasi-reflexive, quasi-symmetric, quasi-transitive and quasi-substitutive only hold for the strong equality (even potentially undefined values can be substituted). Therefore, the strong equality is of outstanding importance for deduction.

Also, consider for example the following operation specification:

```
context C::m(a:Integer):Integer
post: result = 5 div a
```

What is the semantics of this operation given that the precondition does not rule out `a=0`? If the standard strict equality is used this results in an inconsistent specification. If the strong equality is used this operation simple returns undefined when called with an argument of `0`. Depending on the circumstances, both may be reasonable. Thus we suggest to extend OCL with a strong equality operation.

4.6 Recursion

The OCL standard vaguely requires that recursions should always be terminating to rule out problems with divergent operation invocations:

> The right-hand-side of this definition may refer to operations being defined (i.e., the definition may be recursive) as long as the recursion is not infinite. *(OCL Specification [24], page paragraph 7.5.2, pp.16)*

and also:

> For a well-defined semantics, we need to make sure that there is no infinite recursion resulting from an expansion of the operation call. A strict solution that can be statically checked is to forbid any occurrences [...]. However, allowing recursive operation calls considerably adds to the expressiveness of OCL. We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable. *(OCL Specification [24], page A-31)*

Unfortunately, in a proof-environment we have to be substantially more specific than this. Furthermore, HOL-OCL is designed to live with the open-world assumption, i.e., with the potential extensibility of object universes, as a default; further restrictions such as finalizations of class diagrams or a limitation to Liskov's Principle [20] may be added on top, but the system in itself does not require them. This has the consequence that even in the following example:

```
context C::m(a1:T1,...,an:Tn):Integer
  post: result = if a1.p()
                 then 1 + self.m(a1.q(),...,an)
                 else 0 endif
```

the termination for the invocation `self.m(a1.q(),...,an)` is fundamentally unknown (even if `p` and `q` are known and terminating): a potential overriding may destroy the termination of this recursive scheme.

In form of a pre-translation process, operation specifications with a limited form of recursive invocations can be converted into a format that satisfies the constraints of a finite family of constant definitions. These limited forms can be listed as follows:

- calls to superclass operations, i.e., `(self.asType(A)).m(x1,...,xn)`, or
- direct recursive well-founded invocations, i.e., `(self.asType(C)).m(x1,...,xn)` where the user specifies a "measure" or a well-founded ordering which the system checks to be respected in all calls.

The first can be statically resolved, the latter is based on the theory of well-founded orders and the well-founded recursor "wfrec" in Isabelle/HOL, and so to speak an application of the standard HOL methodology to OCL.

Alternatively, in case of a finalized class, i.e., a class that cannot be further extended by inheritance, late-binding can be can by replaced by the case-switch:

$$\text{if } \mathit{self}\texttt{->IsType}(A) \text{ then } S \text{ else if } \ldots \text{ else } S''$$

Summing up, conservativity implies that only limited forms of recursive invocations are admissible. In an open world (no class finalization so far), only operation invocations on objects can be treated, whose type has been fixed, in a closed world (the class hierarchy has been finalized), an invocation can be expanded to a case-switch considering the dynamic type of *self* over calls.

5 The Future

Future extensions will aim for allowing smooth transitions from specification to code ("methods" implementing "operations" in the terminology of the standard). There are various aspects of this global challenge, which are in parts already discussed in research communities as well as in the standardization process.

5.1 Library Extensions

In more recent draft-versions of the standard, *bounded* versions of Integers were suggested. If the purpose of these types is to allow a transition to implementation

languages, then we suggest to choose also a concrete machine arithmetic based on a two's complement model (as described in [11]; similar concrete definitions for C++ are in preparation). These machine arithmetics are somewhat more distant to mathematics (MaxInt + 1 = MinInt, all commutative ring properties hold except $a - a = 0$, $0 \leq \text{abs}(a)$ does not hold even for defined a, etc.) but close to widely used implicit standards in microprocessor technology. Verification of such transitions from mathematical integers to machine integers can be a real concern in safety-critical applications. For a concrete proposal of a "strong" formalization of the Java arithmetics, see [26].

It is conceivable to drop the standards limitation to *finite* collection types. Infinite sets are clearly a very powerful and useful standard means which would allow to explicitly use the infinite sets occurring implicitly in OCL (such as the set of Integers), e.g., by quantifying over them. But there are also useful applications for infinite sequences and bags: They pave the way for new forms of recursion. A recursive method over an object-structure collecting the sequence of values of an attribute does not necessarily have to terminate: semantics could be defined via co-recursion. For code-generators, this means that lazy evaluation techniques known from functional programming must be applied.

5.2 References and Referential Types

One way to view objects with attributes $a_1, \ldots a_n$ of OCL types $T_1, \ldots, T_n$ are tuples of type $\tau_1 \times \ldots \times \tau_n$ where τ_i is the corresponding HOL type of T_i[2]. An object universe can therefore be constructed as the sum over all Cartesian products representing objects. The state of a system is then a partial map from object-id's (*oid*'s) to the object universe.

We suggest one little change of this scheme: the oid should be encoded into the Cartesian products as well, comparable to a "hidden attribute": $\text{oid} \times \tau_1 \times \ldots \times \tau_n$, and states should be restricted to contain only objects whose oid points to itself in the state. This invariant is easy to implement by constructors in an object-oriented programming language: just generate the fresh oid and store it in the object. However, this slight change makes the logical equality $\_ \triangleq \_$ coincide with the referential equality used in many programming languages—as long as we are comparing objects within one state. This makes the formal model of sets, for example, which implicitly uses this equality, much more realistic. This extension of the object encoding scheme is called *referential (object) universe* in [6].

Moreover, this extension also has advantages for programming languages used for implementing methods (see IMP++ for example in [7], which also contains a verification technique). For example, having a reference attribute in each object greatly simplifies the semantic definition of a reference operator (like e.g., `&x` in C++, mapping the type of `x` to its reference type). Further, the assignment operator assigning a reference to an attribute of type oid can then also

[2] We ignore the complications resulting from extensible subtyping here; see [6] for details

be realized easily. Referential universes also give a possible interpretation of the "null-objects" mentioned in the recent versions of the standard "ptc/06-05-01".

5.3 Frame-Properties

The OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i.e., it allows arbitrary relations from pre states to post states. For most applications this is too general: there must be a way to express that parts of the state *do not change* during a system transition, i.e., to specify the frame properties of system transition. Thus we suggest to extend OCL to specify the frame properties explicitly:

```
(S:Set(OclAny))->modifiedOnly():Boolean
```

where `S` is a set of objects (i.e., a set of `OclAny` objects). This also allows recursive operations to collect the set of objects that are potentially changed by a recursive function. Obviously, similar to `@pre` the use of `->modifiedOnly()` is restricted to postconditions.

The definition of the semantics for `->modifiedOnly()` based on the referential universe (see previous section) is straight-forward:

$$X\texttt{->modifiedOnly()} \equiv \lambda(\tau,\tau').\ \llcorner \forall i \notin \left(\text{oidOf} \setminus \ulcorner X(\tau,\tau')\urcorner\right).\ \tau\ i = \tau'\ i \lrcorner$$

where oidOf is just the projection of an object to its *oid*. By the projection, the object set X represents a set of references to values in the store. All objects with *oid*'s not occurring in the set are assumed to be unchanged; for *oid*'s occurring in the set, nothing is specified. Thus, requiring `Set{}->modifiedOnly()` in a postcondition of an operation allows for stating explicitly that an operation is a query in the sense of the OCL standard, i.e., the `isQuery` property is true.

In contrast to frame properties in JML [19], which allow for specifying *attributes* to be assignable or not, our mechanism is a *semantic* and not a *syntactic* one that just forbids assignments to certain paths. This solves the alias problem: `Set{self.a}->modifiedOnly()` is just equivalent to `Set{self}->modifiedOnly()` iff `self.a` and `Set{self}` denote the same object.

6 Conclusion

In our view, there is the need to complement the UML/OCL standardization process by continuous efforts to find a formal semantics.

Ideally, this should be a machine-checked semantics like [6] that might become part of the standard document. As can be seen by similar standardization processes (as, for example, the ISO standardization process of the Z language [17]), such a "beau ideal" semantics has the advantage to turn UML into a real formal method with its potential for high-quality analysis and verification tools. The latter paves the way for light-weight approaches such as [12] for large-scale

industrial applications as well as more heavy-weight system verifications satisfying even EAL7 certification levels ("Formally Verified Design and Tested") of the Common Criteria international standard (ISO/IEC 15408).

It can be safely stated that in contrast to the wealth of informal papers on OCL semantics, a machine-checked semantics results in a higher degree of completeness and perfection. Its main advantage is that it can be used to build an *integrated* semantics, covering data-oriented specification, behavioral specification and programming-language like facets of the UML. Thus, different stakeholders in the standardization process could provide an extension of their proposed UML extension by an extension of the current version of the "beau ideal" semantics to see if their proposed features are in fact consistent with the language. Building such an integrated semantics by paper-and-pencil reasoning or by a design-by-committee process is doomed to failure in the light of our experience.

On the other hand, each attempt to build a formal semantics also results in a certain inflexibility—a lesson that can also be learned from the Z standardization process. This holds to an even larger extent if the semantic representation is machine-checked, requiring that at least representatives of the various stakeholders have sufficient technical skills to handle the underlying theorem prover technology. Similar to standardization efforts centered around a reference implementation, a slow-down of the process is inevitable the more features have been added to the language.

Admittedly, starting the semantics formalization process too early can kill the standardization process as a whole. Not starting it at all, or remaining in a state where only partial approaches exist, will result in a huge inconsistent piece of IT literature. Finding the right balance between informal requirements capture and formalization efforts in the semantics and finding the right point in time to make formal semantics more mandatory in the UML standardization process will therefore be, in our view, crucial for the long-term success of the UML in the future.

Acknowledgement:

We thank the anonymous referees whose remarks helped to clarify this paper.

References

1. The HOL-OCL website, 2006. `http://www.brucker.ch/research/hol-ocl/`.
2. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.
3. A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.
4. A. D. Brucker and B. Wolff. HOL-OCL: Experiences, consequences and design choices. In J.-M. Jézéquel, H. Hussmann, and S. Cook, eds., *UML 2002*, no. 2460 in LNCS, pp. 196–211. Springer, 2002.

5. A. D. Brucker and B. Wolff. Using theory morphisms for implementing formal methods tools. In H. Geuvers and F. Wiedijk, eds., *Types for Proof and Programs*, no. 2646 in LNCS, pp. 59–77. Springer, 2003.
6. A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zürich, 2006.
7. A. D. Brucker and B. Wolff. A package for extensible object-oriented data models with an application to IMP++. In A. Roychoudhury and Z. Yang, eds., *SVV 2006*, Computing Research Repository (CoRR). 2006.
8. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
9. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam Manifesto on OCL, 1999.
10. M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.
12. B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pp. 232–241. ACM Press, 2006.
13. R. Hähnle. Towards an efficient tableau proof procedure for multiple–valued logics. In E. Boerger, H. K. Buening, M. M. Richter, and W. Schoenefeld, eds., *CSL 90*, *LNCS*, vol. 533, pp. 248–260. Springer, 1991.
14. A. Hamie, F. Civello, J. Howse, S. Kent, and M. Mitchell. Reflections on the Object Constraint Language. In *Post Workshop Proceedings of UML 98*. Springer, 1998.
15. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, pp. 64–72, 2004.
16. R. Hennicker, H. Hussmann, and M. Bidoit. On the precise meaning of OCL constraints. In T. Clark and J.Warmer, eds., *Advances in Object Modelling with the OCL*, *LNCS*, vol. 2263, pp. 69–84. Springer, 2002.
17. Formal Specification – Z Notation – Syntax, Type and Semantics. 2002. Draft International Standard.
18. S. N. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA 86*, pp. 406–416. ACM Press, 1986.
19. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, eds., *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers, 1999.
20. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. 16(6):1811–1841, 1994.
21. L. Mandel and M. V. Cengarle. On the expressive power of OCL. In *FM 99*. 1999.
22. Object constraint language specification. 1997. Covers OCL 1.1.
23. OMG Unified Modeling Language Specification. 1999. Covers UML/OCL 1.3.
24. UML 2.0 OCL specification. 2003. `ptc/2003-10-14`.
25. OMG Unified Modeling Language Specification. 2003. `formal/03-03-01`.
26. N. Rauch and B. Wolff. Formalizing Java's two's-complement integral type in isabelle/HOL. In *ENTCS*, vol. 80. Elsevier Science Publishers, 2003.
27. M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Ph.D. thesis, Universität Bremen, 2002.
28. M. Vaziri and D. Jackson. Some shortcomings of OCL. Response to Object Management Group's Request for Information on UML 2.0.
29. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

Improving the OCL Semantics Definition by Applying Dynamic Meta Modeling and Design Patterns

Juan Martín Chiaradía [1] Claudia Pons [1,2]

[1] LIFIA – Facultad de Informática, Universidad Nacional de La Plata

[2] CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)

La Plata, Buenos Aires, Argentina

{jmchiara,cpons}@sol.info.unlp.edu.ar

Abstract. OCL is a standard specification language, which will probably be supported by most software modeling tools in the near future. Hence, it is important to OCL to have a solid formal foundation, for its syntax and its semantic definition. Currently, OCL is being formalized by metamodels expressed in MOF, complemented by well formedness rules written in the own OCL. This recursive definition not only brings about formal problems, but also puts obstacles in language understanding. On the other hand, the OCL semantics metamodel presents quality weaknesses due to the fact that certain object-oriented design rules (patterns) were not obeyed in their construction. The aim of the proposal presented in this article is to improve the definition for the OCL semantics metamodel by applying GoF patterns and the dynamic metamodeling technique. Such proposal avoids circularity in OCL definition, and increases its extensibility, legibility and accuracy.

Keywords: OCL; formal semantics; dynamic meta modeling; design patterns.

1 Introduction

OCL (Object Constraint Language) [8] is a formal specification language, easy to read and write, accepted as a standard by the OMG (Object Management Group). OCL permits to define syntactic and semantic restrictions upon models expressed in graphic notations such as the UML [13], thus extending the expressive capacity of such notations. In this way, diagrams complemented by OCL expressions are more accurate and complete.

Both UML and OCL are defined by MOF (Meta Object Facility) [6], which is a meta-language maintained by OMG whose aim is to allow for metamodel creation.

The OCL language has been formally defined through the following documents:

- a MOF (meta) model that defines its abstract syntax.
- a MOF (meta) model that describes its semantic domains.
- a set of MOF classes that specify the OCL semantic (meaning), i.e. the connection between the syntactic constructions and the semantic domain.

It is known that the object-oriented models, due to their proximity to reality, transmit an intuitive meaning, easy to be perceived by their readers; however, when the design of such models is not adequate, intuition disappears, and models become difficult to understand. This unfavorable situation is observed in some parts of the OCL 2.0 standard specification [8]. The reason why this occurs may be the non-application (or inadequate application) of some well-known design patterns. Although in the abstract syntax definition the result obtained is clear and accurate, in the semantic definition several questions arise that hamper language understanding. We believe that this lack

of “self- explaining” observed in [8] is due to an erroneous selection of the design patterns used in the design of the semantics metamodel.

Working towards the solution to this problem, we propose to create a clearer and simpler alternative definition for the OCL semantics. For that purpose, GoF patterns [4] will be applied to the design presented in the standard specification document [8]. Our hypothesis is that pattern application will contribute to improve legibility, extensibility and accuracy of OCL definition.

To provide an adequate context to the reading of this proposal, in Section 2 we present a summary of the current OCL semantics [8]. Then, in Section 3, we propose a new definition for the OCL semantics, based on the *Visitor* pattern application [4] upon the semantics metamodels; we also use the technique known as *Dynamic Meta Modelling* (DMM) [2] [5] to achieve an accurate specification of the semantics, but keeping clarity, and communicating concepts in a more intuitive manner. Finally, in Section 4, we present conclusions and future works.

2 OCL Specification Overview

OCL expression is defined in [8] as "an expression that can be evaluated in a given environment" and it states that “evaluation of the expression yields a value". Taking it into account, the ‘meaning’ (semantics) of an OCL expression can be defined as the value yielded by its evaluation in a given environment. In order to specify this semantics, [8] proposes the structure illustrated in figure 1.

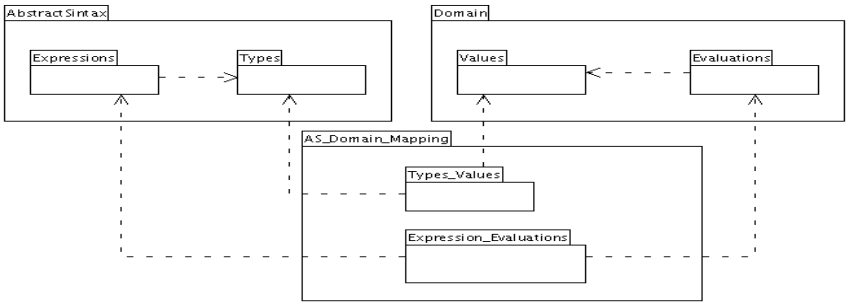


Figure 1: Overview of packages in the UML-based semantics

Figure 2 shows the overview of the *AbstractSintax* package, which defines the abstract syntax of OCL as a hierarchy of meta classes. In the other hand, *Evaluations* package defines the semantics of these expressions using also a hierarchy of meta classes where each one represents an evaluation of a particular kind of expression (see figure 3). The idea behind this representation is that each evaluation yields a result in a given environment, therefore, the semantics evaluation of an expression in a specific environment is given by associating each evaluation instance to an expression model (see figure 4).

It is easy to see how *Evaluations* package replicates the hierarchy of the abstract syntax. We believe that this duplication is unnecessary and yields to disadvantages such as low legibility of the meta model and inefficiency in the development of automatics tools based on this semantics. We will expand this in the next section.

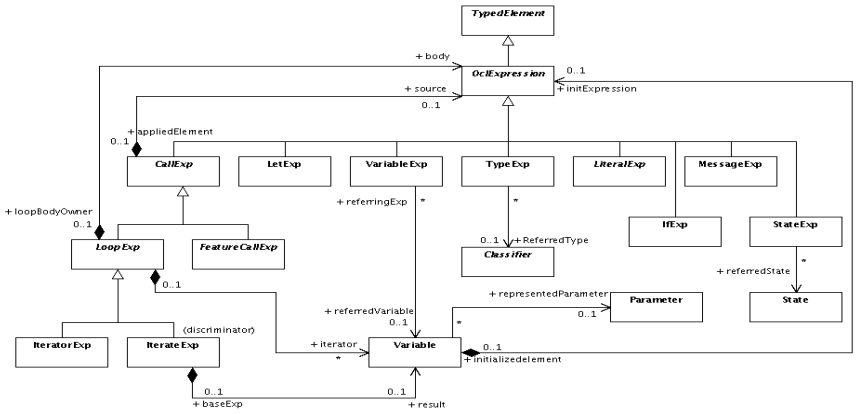


Figure 2: *AbstractSyntax* package overview

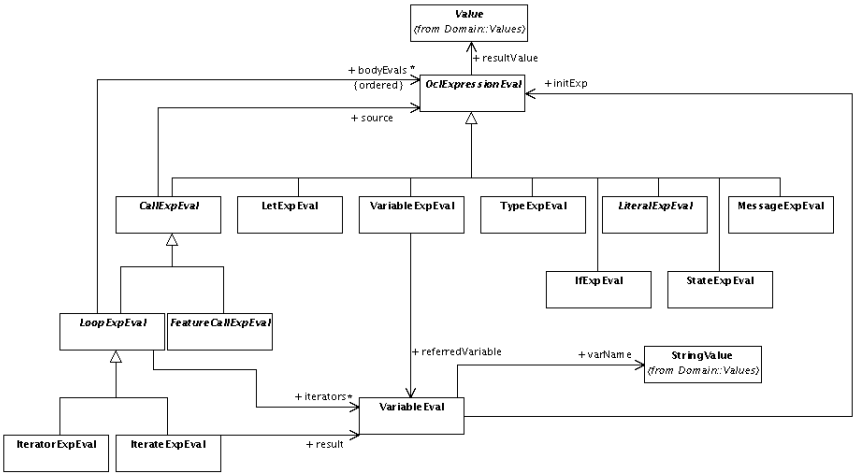


Figure 3: *Evaluations* package overview

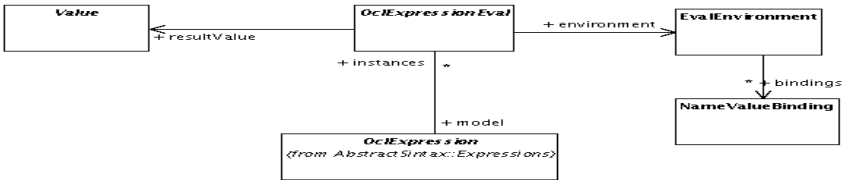


Figure 4: Semantics Evaluation of an expression.

3 Semantics evaluation through "Visitor" pattern and DMM

In this section we define a meta class named *OclEvaluator* to give semantic meaning to syntax expressions by associating them with its corresponding value. In this way, *OclEvaluator* works as a bridge between AbstractSyntax and Values packages (see figure 5).

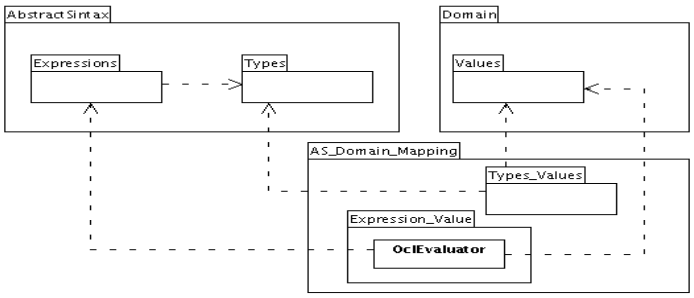


Figure 5: OCL meta model using the Visitor

In order to evaluate an expression, the *OclEvaluator* uses an evaluation environment called *EvalEnvironment* following the classic strategy used in semantic definition of programming languages (examples of this approach can be found in [3] and [10]). An expressions evaluation depends on its evaluation environment as well as its syntax structure.

The *OclExpression* structure is not likely to change, and several operations might be defined (e.g. refactoring operations, semantics evaluation, code generation operations, etc.). Consequently, we consider that it is more appropriate to avoid polluting the static structure with these operations and then to apply the Visitor pattern [4], in order to keep it simple and clear (See figure 6).

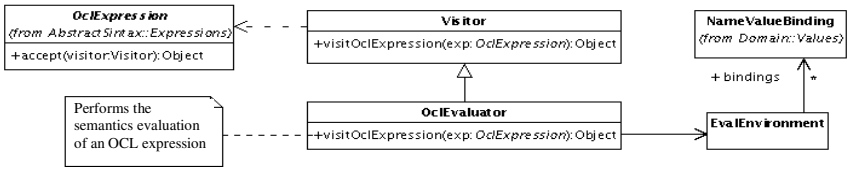


Figure 6: Evaluation metamodel using the Visitor pattern.

In addition, we believe that the best way to understand the semantics evaluation is by showing the evaluation process itself. By using only class diagrams to reflect the semantics evaluation, it is hard (or almost impossible) to reveal the latter process, because of the static nature inherent to these diagrams. Furthermore, to completely understand all the process it is necessary to pay attention to the constraints established on these diagrams. In [8], these constrains are written in OCL with two negatives outcomes:

- The expressibility and simplicity obtained from the use of UML in the semantics metamodel over the math one is lost because of the necessity of be aware of the constraints to fully understand the semantic.
- The constraints are written in OCL, so that the semantics of OCL is defined in terms of OCL itself! If someone didn't understand OCL, they would neither understand these constraints.

Consequently, with the aim of a simple, precise and clear explanation, in this section we use sequence diagrams to visualize the distinct steps throughout the semantics evaluation of expressions. This approach is known as *Dynamic Meta Modelling* (DMM) [2] [5], and has been used in the semantics specification of UML elements (such as State Machines and Collaborations), but its use in OCL specification has not been explored before.

3.1 *Semantics of a LetExp*

The evaluation of a *LetExp* proposed in [8] is shown in figure 7. The diagram shows how the evaluation encapsulates the result value and the evaluation environment, although neither the evaluation method nor structural constraints are specified on this diagram.

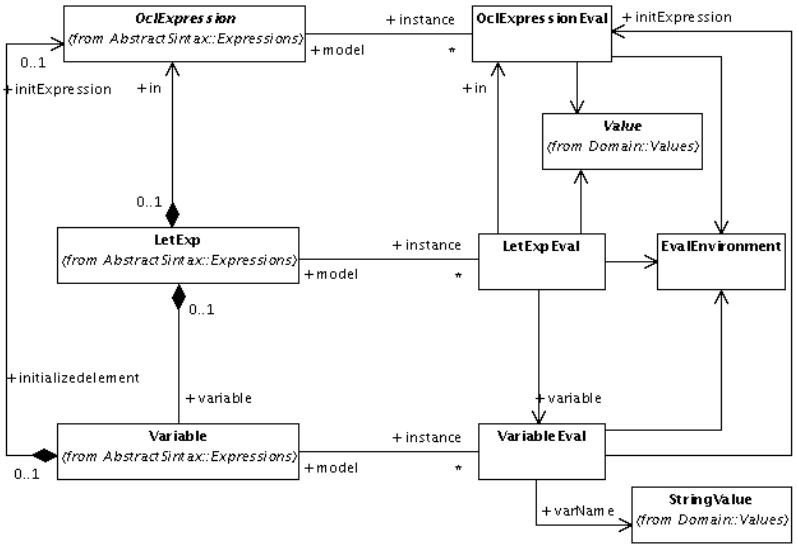


Figure 7: Standard UML based semantic evaluation of a *LetExp*.

Therefore a simple analysis of the last diagram doesn't give us too much information about the semantics of a *LetExp*; it only gives us information about the static structure of the elements implied in this evaluation. In order to fully understand the previous diagram, we must study its well formed rules [8].

As we previously established, these constraints have the disadvantage that they are written in OCL, which clearly becomes an obstacle for those who give their first steps in OCL.

The appendix A of [7] presents the maths model of the OCL semantics (see figure 8). Taking it into account, we can translate this algorithm under the applicative order reduction into a sequence diagram (see figure 9).

A context for evaluation is given by an environment $\tau = (\sigma, \beta)$ consisting of a system state σ and a variable assignment $\beta: Var_t \rightarrow I(t)$. A system state σ provides access to the set of currently existing objects, their attribute values, and association links between objects. A variable assignment β maps variable names to values.
Let *Env* be the set of environments $\tau = (\sigma, \beta)$. The semantics of a *LetExp* is a function $I[e]:Env \rightarrow I(t)$ that is defined as follows.

$$I[\![\text{let } v = e_1 \text{ in } e_2]\!](\tau) = I[\![e_2]\!](\sigma, \beta\{v/I[\![e_1]\!](\tau)\}).$$

Figure 8: Maths **s**emantics of *LetExp*

As a first step of evaluation, we evaluate the *init* expression ($I[\![e_1]\!](\tau)$, signals 4 and 5) to extend the evaluation environment with the latter evaluation ($\beta\{v/I[\![e_1]\!](\tau)\}$, signals 6, 7 and 8). Then, we evaluate the *in* expression in the new environment, and the value returned by this is the result of the whole *LetExp* evaluation

($I[\![e_2]\!](\sigma, \beta\{v/I[\![e_1]\!](\tau)\})$, signals 9, 10 and 11). Note that the internal environment modification were propagated outside the *LetExp* evaluation, we saved the environment at the beginning of the evaluation process to recover it when the evaluation is finished (signals 2 and 12).

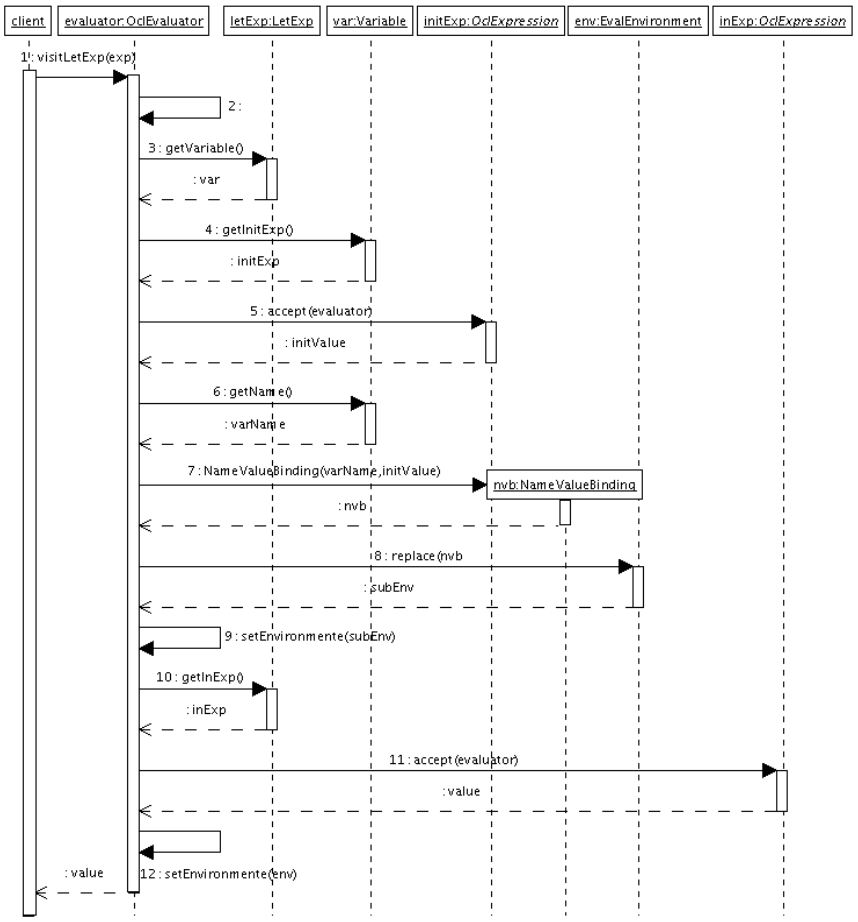


Figure 9: Sequence diagrama of a *LetExp* evaluation.

3.2 *Semantics of IterateExp*

The semantics evaluation of an *IterateExp* as is expressed in [8] is shown in figure 10. Once again we have the problem that the chart doesn't express too much about the semantics evaluation process and we have to appeal to the well formedness rules established on this diagram [8]; without these constraints we would be unable to completely understand the semantic process of an *IterateExp.*

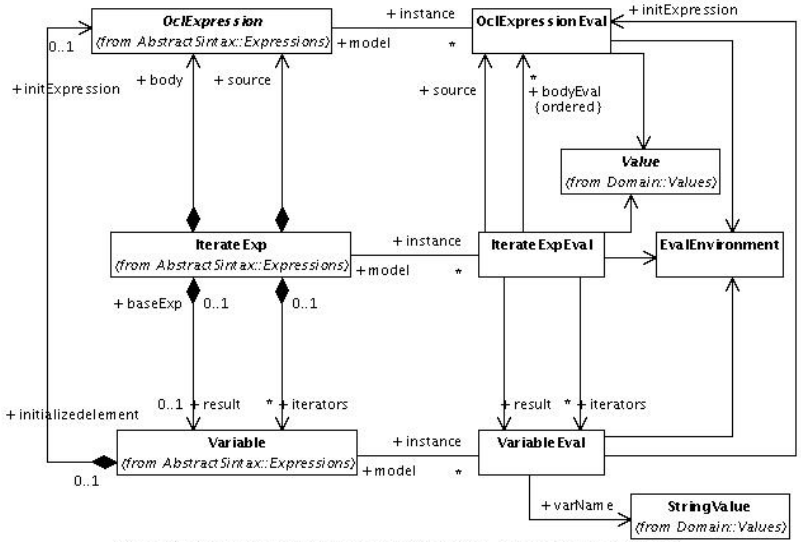


Figure 10: Standard UML based semantic evaluation of an *IterateExp.*

Even worse, such constraints try to explain how *IterateExp* works but lacks of correctness due to the fact that the *IterateExp* is defined in terms of a *ForAllExp* wich is itself defined in terms of *IterateExp*, as follows:

> The environment of any sub evaluation is the same environment as the one from its previous sub evaluation, taking into account the bindings of the iterator variables, plus the result variable which is bound to the result value of the last sub evaluation.

```
context IterateExpEval inv:
let SS: Integer = source.value->size()
in if iterators->size() = 1
then Sequence{2..SS}->forAll(i:Integer | bodyEvals-
>at(i).environment = bodyEvals->at(i-1).environment-
>replace(NameValueBinding(iterators->at(1).varName,
source.value->asSequence()->at(i)))-
>replace(NameValueBinding(result.varName,bodyEvals->at(i-
1).resultValue )))
else -- iterators->size() = 2
Sequence{2..SS*SS}->forAll(i: Integer | bodyEvals-
>at(i).environment = bodyEvals->at(i-1).environment->replace(
NameValueBinding( iterators->at(1).varName,source-
>asSequence()->at(i.div(SS) + 1)))->replace(
NameValueBinding( iterators->at(2).varName,source.value-
>asSequence()->at(i.mod(SS))))->replace(
NameValueBinding(result.varName,bodyEvals->at(i-
1).resultValue )))
endif
```

Although an *IterateExp* is more complicated than a *LetExp,* without a previous OCL knowledge, it is almost impossible to understand these constraints, and with the proper knowledge of the language, the reading and comprehensiveness of these constraints is a hard task to do.

As we have done whit the *LetExp*, we use the math semantics of the *IterateExp* as guidance for showing this process through a sequence diagram. A summary of the math semantics is shown in figure 11 (see Appendix A of [7] for the full version), while figure 12 and figure 13 display the semantics expressed via sequence diagrams.

$I[\![e_1 \to \text{iterate}(v_1; v_2 = e_2 \mid e_3)]\!](\tau) = I[\![e_1 \to \text{iterate}'(v_1 \mid e_3)]\!](\tau')$ where $\tau' = (\sigma, \beta')$ and $\tau'' = (\sigma, \beta'')$ are environments with modified variable assignments

$$\beta' := \beta\{v_2 / I[\![e_2]\!](\tau)\}$$
$$\beta'' := \beta'\{v_2 / I[\![e_3]\!](\sigma, \beta'\{v_1 / x_1\})\}$$

and it

(a) If $e_1 \in \text{Expr}_{Sequence(t_1)}$ then

$$I[\![e_1 \to \text{iterate}'(v_1 \mid e_3)]\!](\tau') = \begin{cases} I[\![v_2]\!](\tau') \\ \quad \text{if } I[\![e_1]\!](\tau') = \langle\rangle, \\ I[\![\text{mkSequence}_{t_1}(x_2, \ldots, x_n) \to \text{iterate}'(v_1 \mid e_3)]\!](\tau'') \\ \quad \text{if } I[\![e_1]\!](\tau') = \langle x_1, \ldots, x_n \rangle. \end{cases}$$

Figure 11: Maths semantics of IterateExp

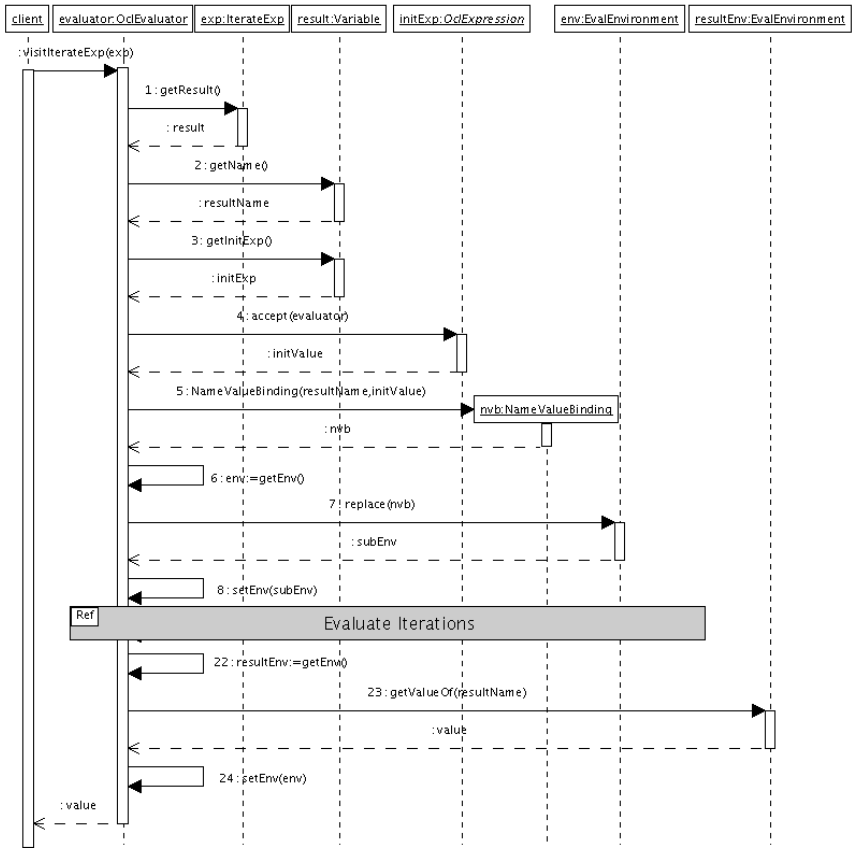


Figura 12: *IterateExp* semantics as sequence diagram.

IterateExp evaluation can be seen as follows:

The first sub evaluation will start with an environment in which the result variable is bound to the init expression of the variable declaration in which it is defined ($\beta' := \beta\{v_2 / I[\![e_2]\!](\tau)\}$, signals 1 to 8 in figure 12); then we proceed to evaluate the body with all iterator variables bound to the different combinations of the source (figure 11). The iterators binding ($\beta'\{v_1/x_1\}$ in $\beta'' := \beta'\{v_2 / I[\![e_3]\!](\sigma, \beta'\{v_1/x_1\})\}$) is done by *CombinationGenerator* (signals

13, 14 and 15 in figure 13), under a 'depth first search' strategy. This strategy determines the number of sub evaluations over the body ($I[\![e_3]\!](\sigma,\beta'\{v_1/x_1\})$ in $\beta'' := \beta'\{v_2/I[\![e_3]\!](\sigma,\beta'\{v_1/x_1\})\}$; signals 17 and 18 in figure 13); as last step, these sub evaluations will update the result variable ($\beta'\{v_2/I[\![e_3]\!](\sigma,\beta'\{v_1/x_1\})\}$,signals 19, 20 and 21 in figure 13).

Once again we save the environment at the beginning of the evaluation process and, after recovered the value bound to the result variable (signals 22 and 23 in figure 12), we restore the initial environment.

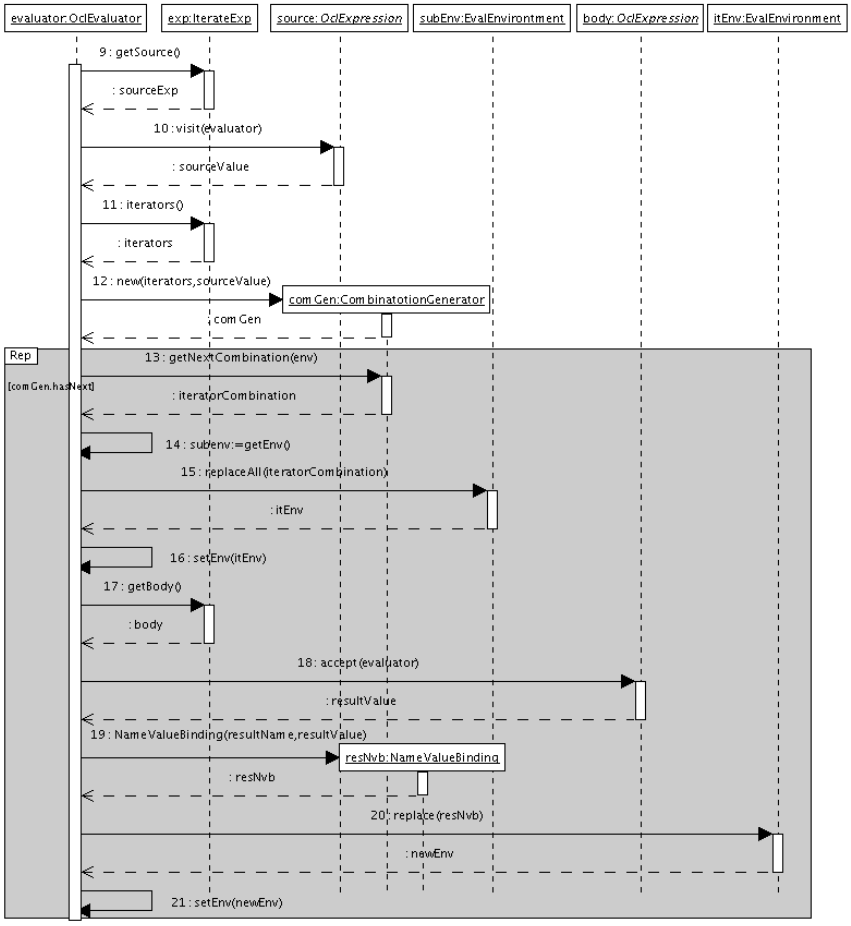


Figure 13: Body Evaluation of an *IterateExp.*

With this approach, each meta-class belonging to the Domain package will be replaced with a sequence diagram which states the concrete semantics and evaluation process of the corresponding syntactic construction.

4 Conclusion and Future Works

OCL is an object property specification language, which is rigorous but simple and easy to use. Therefore, it becomes a very interesting option for the development of

code verification and derivation tools. To exploit all its potential, it is fundamental that OCL has a solid formal foundation for both its syntax and its semantic definitions. The OCL standard is formalized by metamodels expressed in MOF, complemented by well formedness rules written in the own OCL. This circular definition not only gives rise to formal problems [11], but also puts obstacles in language understandings. Additionally, we think that the use of (static) meta-classes to express the OCL semantics was a wrong choice, because of the dynamic nature of semantics evaluation which requires a dynamic (meta) modeling tool.

In this article, we elaborate an alternative definition for the OCL semantics. This proposal re-uses the OCL syntax metamodel, re-designs the OCL semantics metamodel by applying the 'Visitor' design pattern, and finally defines the relation between syntax and semantics through UML collaboration diagrams adhering to the DMM approach. In this way, circularity on the OCL definition is avoided, and intuitive communication is increased. Besides, the OCL math semantics was used as a foundation and guidance for the semantics definition. Although math semantics could be tedious and hard to understand, and demands users with more academic background, we showed that it could be translated into sequence diagrams offering a more readable and simple semantics metamodel.

On the other hand, the adequate performance of the tools supporting OCL [12] [1] strongly depends on the quality of language definition. To count on a well-defined syntax and semantics will result in benefits for such tools. Also, it is almost straightforward to translate this semantics into a programming language such as Java, because of the proximity between sequence diagrams and programming languages.

Finally, the application of the visitor pattern makes it easier the creation of new functionality over the OCL syntax structure and its integration into the CASE tool to get a powerful one. For example, concerning model transformations, it is possible to define OCL constraints transformations by adding a new "visitor" for the OCL syntax hierarchy. In this sense, we are working on the redefinition of the ePlatero evaluator [9] following the proposal presented in this article in order to analyze the potential advantages regarding the different indicators, such as reliability, efficiency, modifiability, etc.

References

[1] Akehurst David: "OCL 2.0 – Implementing the Standard for Multiple Metamodels" - URL: http://www.cs.kent.ac.uk/projects/ocl/Documents/OCL%202.0%20-%20Implementing%20the%20Standard.pdf

[2] Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: "Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml". In Evans, A., Kent, S., Selic, B., eds.: UML 2000, York, UK, October 2-6, 2000, Proceedings. Volume 1939 of LNCS, Springer (2000) 323–337

[3] Hennessy, M.: "The Semantics of Programming Languages: An elementary introduction using structural operational semantics". J Wiley&Sons. England. 1990.

[4] Gamma, E. Helm, R. Johnson, R. and Vlissides, J.: "Design Patterns, Elements of Reusable Object-Oriented Software". Addison-Wesley Publishing Company, 1995.

[5] Hausmann, J.H.: "Dynamic Meta Modeling. A Semantics Description Technique for Visual Modeling Languages" PhD thesis, Universit¨at Paderborn, Germany (2005)

[6] Meta Object Facility MOF 2.0. OMG Adopted Specification ptc/2003-10-04. URL:www.omg.org

[7] OCL 2.0.– OMG draft Specification /ptc/03-10-14. URL: www.omg.org/docs/ptc/03-10-14.pdf

[8] OCL 2.0 Specification ptc/2005-06-06. URL:www.omg.org

[9] Pons, Claudia, R.Giandini, G. Pérez, P. Pesce, V.Becker, J. Longinotti, J.Cengia. “Precise Assistant for the Modeling Process in an Environment with Refinement Orientation” In "UML Modeling Languages and Applications: UML 2004 Satellite Activities". Lecture Notes in Computer Sciefnce number 3297. Springer-Verlag. 2004. ISBN: 3-540-25081-6

[10] Reynolds, John C.: “Theories of Programming Languages” Cambridge University Press.

[11] Tchertchago Alexei: “Analysis of the Metamodel Semantics for OCL”. URL: http://www.hwswworld.com/downloads/9_28_05_e/Cherchago-thesis.pdf

[12] Richters Mark and Gogolla Martin. “OCL-Syntax, Semantics and Tools” in Advances in Object Modelling with the OCL Lecture Notes in Computer Science 2263. Springer. (2001).

[13] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification.. http://www.omg.org. August 2003. URL:www.omg.org

Sugar for OCL

Jörn Guy Süß

Information Technology and Electrical Engineering
The University of Queensland, St. Lucia, 4072, Australia
jgsuess@itee.uq.edu.au

Abstract. Examples of OCL use often do not exceed a few lines. Larger examples are rare, because the concrete syntax of OCL is verbose and based exclusively on ASCII encoding. This makes it easy to edit OCL in any environment, but hard to layout in a readable manner. A minor issue like presentation affects use in a major way. This paper proposes three shorthand notations, or syntactic sugars, for laying out OCL in the Latex, HTML, and Unicode encoding systems. To avoid splitting the available OCL source code base any further, flavours are convertible via the base syntax. To allow benefit across the community, the representations are OCL version-independent. To support recognisability, the representations are visually very similar. To simplify reuse, definitions are based on POSIX regular expressions and Unicode.

1 Introduction

While OCL today offers an substantial number of tools, its adoption as an industry standard is still limited. Usage issues due to language semantics and tooling, like the lack of modularisation, non-deterministic evaluation, the missing and insufficiently formalized transitive closure operation and collection flattening have been addressed successfully in the past[2,**?**] and seem to consolidate. OCL's concrete syntax still seems to be an impediment:

Working with a UN-CEFACT group on the metamodel for the business language UMM[8], which is formalized as a UML profile, I had to introduce the workgroup to the use of OCL. It turned out that even simple examples quickly filled the whiteboard. Points of emphasis were hard to make, due to the expansive syntax. Thus I resorted to short symbols, borrowed from math, logic or improvised in the process, but made clear that these were *not* standard compliant. After collecting the workshop notes, I found that most participants had adopted the shortened ad-hoc syntax and that those that had used it, had generally tried out more and different formalisation solutions, and hence come up with better ones on average, then those that had used the standard syntax.

After this experience, I applied the shorthand to the OCL contained in my work of creating a UML Profile for small-scale enterprise integration, to save print space. Section 4 shows two Well-formedness Rules from the UML 1.4.2

standard. I then showed parts of the work to some colleagues who had previously criticised OCL as 'to bulky' and 'not mathematical', to find increased acceptance, purely because of a syntactic sugar.

Trying to apply this encouraging result, some underlying challenges and requirements surfaced. Foremost, OCL is not a single language, which conforms to a single grammar or meta-model, but a collection of languages with an overlapping concrete syntax. The published grammar in the UML 1.4.2 standard for example cannot be directly converted into an LALR1 parser[12], which prompts several dialects. The OCL 2.0 standard, although a great improvement, only contains a non-normative concrete syntax section. Consequently, the use of a parser to analyse source code is not very promising. Most OCL tools however adhere to keyword and syntax conventions laid down in OCL 1.6 and to the standard library of collection functions shown there. This proposal thus uses OCL token patterns, rather than full parser analysis. This allows its application even to unstructured or broken OCL code, which broadens applicability and robustness of the approach. A technically and semantically viable form to specify such 'approximate matches' is the use of regular expressions (REs). REs are an established means to parse constructs from a token stream and have been formalized, standardized and implemented to the degree of commoditisation.

The overall approach involves three steps: Determining *what to abbreviate*, *choosing a set of glyphs* or symbols as abbreviations, which is both intuitive and available within the different technical systems used to display and print OCL, and determining *how to find* matching constructs. The paper is structured as follows. The next section introduces the abbreviation syntax, treating both the keyword selection and glyph system. Section 3 describes the portable technical specification using REs, and section 5 investigates related approaches. Section 6 presents an outlook.

2 Abbreviating OCL

An abbreviation is a mapping from the range of OCL texts using keywords to the domain of OCL texts using symbols. In section 2.1, we choose a common symbol set for the domain and discuss some typeface conventions. The subsequent sections 2.2 to 2.7 discuss the range of abbreviations for structural parts, collections, enumerations types, simple and collection operations, followed, each arguing for the choice of symbol used. As a guideline symbols are introduced for OCL constructs that are mandatory, like the structural parts, or used frequently, like the zero-arity collection operations. Some operations are given shorter textual names. Each section finishes with a summary table.

2.1 Glyph System, Font and Typography

OCL source code intended for reading currently appears in two contexts: As specification text in documents and as part of (UML) models. The notation must cater to these contexts. Academics tend to use the Latex typesetting compiler to author specification documents; in the industrial context Microsoft Word dominates. In addition, specifications are occasionally rendered as HTML documents. Modelling tools these days are either based on Java or directly on the Windows operating system. We thus have to define abbreviations for two glyph systems: Latex and Unicode[1].

To present OCL more like a formula and less like a program, the notation uses a proportional serif font, like 'Times', to typeset all text. This saves space compared with a fixed-width typewriter font. However, serif fonts often do not provide math symbols, so switching of fonts may be necessary. In Microsoft Word for example, the 'Times New Roman' font supplies ASCII glyphs and a few symbols, while the 'MS PMincho' font supplies the complete Unicode 1.4 Mathematical Operators[5] glyph range from hexadecimal 2200-22FF.

The summary tables in the following section give the literal OCL token that is to be replaced, the Unicode abbreviation, symbol number and font, and the Latex code and package, if required. To allow a large degree of portability, the Latex symbols are derived from the mapping of ISO 8879:1986 entities to Latex by Vidar Bronken Gundersen, Rune Mathisen[14]. The leftmost column shows the required latex package or mode. The following sections suggest certain frequently used parts of OCL for abbreviation.

2.2 Structural Parts

OCL source code breaks into packages, which contain context statements holding invariants, definitions or pre- and post-conditions. This structure is abbreviated as follows: Open and closed boxes define the boundaries of a package. The closed box symbol further alludes to the symbol for the end of a mathematical proof. The copyright symbol represents a context. The Greek lowercase lambda represents the 'let' abstraction, as in lambda calculus. Global definitions ('def') are shown as a plus in a box, as they add derived features to the context. The three state restricting stereotypes – invariant, and pre- and post- condition – use a graphic metaphor of a program flow from top to bottom: A rhombus symbol represents an invariant. The symbol alludes to the fact that invariants have to hold before and after a change in the system. The rhombus widens, as invariant conditions are broken and narrows again, as they are restored. Pre- and Post-conditions appear as guards before (above) and after (below) the body of the method, contained in the box. The 'self'-reference of the instance is shown as an arrow reverting to its origin. The summary can be found in Table 1.

[1] Unicode is usable in HTML 4.01, Microsoft Word, the Windows operating system and the Java Virtual Machine.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	Package
package	□	25A1	T	\square	amssymb
endpackage	■	25A0	T	\block	Isoent
context	©	00A9	T	\copyright	-
inv	♢	25CA	T	\diamond	mathmode
let	λ	03BB	T	\lambda	mathmode
def	⊞	229E	M	\boxplus	amssymb
pre		2552	T	\boxDr	Isoent
post		2558	T	\boxUr	Isoent
self	↻	21BB	M	\circlearrowright	amssymb

Table 1. Structural Parts

2.3 Collections

Definition constraints increase the power of OCL through modularisation. This also leads to the usual challenges encountered in object-oriented programming languages [6]. Although the definition of type signatures is optional in OCL, explicit types are an invaluable aid in discovering errors and their use should be encouraged. To this end the lengthy syntax for collection types is abbreviated using three types of braces, common for sets in mathematics, lists in functional programming languages and bags in the Zed[3] notation. The same notation can also be used to express construction of a collection within OCL body text. The summary is found in Table 2.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	Package
Set(X)	{X}	007B / 007D	T	\{ / \}	mathmode
Sequence(X)	[X]	005B / 005D	T	[ ]	mathmode
Bag(X)	⟨X⟩	3008 / 3009	M	\langle / \rangle	mathmode

Table 2. Collections

2.4 Enumerations

Metamodels, like that of the UML, contain enumerations, which often have long names to provide clarity, like VisibilityKind or ChangeableKind. The OCL syntax requires that an enumeration value be declared with the full type and value identifier. With long names, this takes up a lot of space. In fact, within the UML standard's own OCL well-formedness rules the type-name is generally left out. We adopt this simplification and use a typewriter font to mark the enumeration

value as something extraneous to the model. This convention is obviously restricted to models, in which the labels representing the enumeration values are unique. Otherwise a mechanism is needed, which, if provided with the context, returns the intended value to the replacement mechanism.

2.5 Simple Object Operations

Many simple operators have equivalent mathematical symbols, such as the basic Boolean operators, absolute value function and string concatenation.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
<>	≠	2260	T	\not=	mathmode
and	⋀	22C0	M	\wedge	mathmode
or	⋁	22C1	M	\vee	mathmode
not	¬	00AC	T	\lnot	mathmode
implies	⇒	21D2	M	\Rightarrow	mathmode
.abs(x)	|x|	007C	T	\vert x \vert	mathmode
.concat(x)	&(x)	0026	T	\&(x)	

Table 3. Simple Object Operations

2.6 Collection Operations

Different types of arrows distinguish collection operations without parameters from those with parameters. Zero-arity operations are represented by a hook arrow, and leave out the following brace; all other operations are shown with a regular arrow with parameters inside the brace. To shorten the Latex notation further, the operation name is shown atop, rather than behind, the arrow in those notations.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
->x()	↩x	21A9	M	\atop{x}{\hookleftarrow}	mathmode
->x(y)	→x(y)	2192	T	\atop{x}{\rightarrow}(y)	mathmode

Table 4. Simple Object Operations

Collection Operation Names The following operations from the areas of set theory and predicate logic, functional programming and list manipulation, and relational-calculus are used frequently in the definition of well-formedness rules.

The operations for set-theory use customary mathematical symbols. Only the abbreviation for symmetricDifference is a composition. The two last operations -'excluding' and 'including' - use an exclamation mark to indicate that the collection on which the operation is invoked is 'changed'.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
union	$\bigcup$	22C3	M	\cup	mathmode
intersection	$\bigcap$	22C2	M	\cap	mathmode
symmetricDifference	$\bigcup - \bigcap$				mathmode
isEmpty	$\emptyset$	2205	M	\emptyset	mathmode
includes	$\in$	2208	M	\in	mathmode
excludes	$\notin$	2209	M	\not\in	mathmode
forAll	$\forall$	2200	M	\forall	mathmode
exists	$\exists$	2203	M	\exists	mathmode
excluding	$\subset$!	2282	M	\subset	mathmode
including	$\supset$!	2283	M	\supset	mathmode

Table 5. Collection Operation Names

The 'select', 'collect' and 'iterate' operations are equivalents of basic operations from the field of functional programming. The operation 'select' also appears in the Relational calculus, where it is abbreviated as the lowercase Greek letter sigma. This convention is also used here.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
select	σ	03C3	T	\sigma	Mathmode
collect	map		T		Mathmode
iterate	fold		T		

Table 6. Collection Operation Types

The 'count', 'one', 'isUnique' and 'sum' operations often occur in contexts were cardinalities need to be enforced, like in database modelling. The question mark used for the first three is meant to indicate that these are *query* operations, which either query a variable (infix use) or a Boolean property (postfix use).

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
count	|?|	007C	T	\vert ? \vert	mathmode
one	|1|?	007C	T	\vert 1 \vert ?	mathmode
isUnique	key?				
sum	$\sum$	2211	T	\sum	mathmode

Table 7. Database and cardinality operations

Sequence operations are common when building constraints on access structures and in functional programming. The 'any' function is renamed, as it introduces non-determinism, which probably deserves greater recognition.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
append	◁	22B2	M	\triangleleft	mathmode
prepend	▷	22B3	M	\triangleright	mathmode
subsequence	sub		T		mathmode
at	#		T	#	mathmode
any	rnd!		T		

Table 8. Sequence and list operations

2.7 Types, Casts and States

Multiple inheritance and *def*ined additional attributes and operations often require the use of type operators in OCL programs. Unfortunately, the type operators are relatively unwieldy, making it harder to write type-safe operations. We abbreviate the type and kind concepts with Greek lowercase letters Tau and Kappa, followed by a question mark for a predicate and an exclamation mark for a cast. Similarly, the state and creation predicates are shown as a Greek lowercase sigma (end of sentence variant) and nu. These are shown in Table 9.

3 Mapping Mechanism and Strategy

In order for the approach to work, the keywords and syntactic constructs outlined in the previous section cannot be used as variable names, as REs are not aware of the context of occurrence. As all abbreviation patterns are disjunct, the mappings are bijective. Thus, the concrete syntax notation can be used as a pivot to translate, for example, Latex representation to HTML representation. Each mapping is set up as a set of RE search-and-replace pairs.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
oclIsKindOf(X)	κ?	03BA	T	\kappa ?(X)	mathmode
oclIsTypeOf(X)	τ?	03C4	T	\tau ?(X)	mathmode
oclAsType(X)	τ!	03A4	T	\tau !(X)	mathmode
oclInState(X)	ς?	03C3	T	\varsigma ?(X)	mathmode
oclIsNew()	ν?	03BD	T	\nu?	mathmode

Table 9. Types, Casts and States

3.1 Unicode

Translation between concrete syntax and Unicode does not have any issues. Since both glyph systems do not allow font information, the abbreviation of enumerations cannot be performed.

3.2 HTML

HTML 4.01 is used as the translation target standard. Translation from concrete syntax to HTML uses the numeric codes for Unicode entities. To express enumerations, the HTML 'code' tag is used. The advantage of this tag is that it is less presentation oriented, and hence less likely to be affected by presentation mechanisms like Cascading Style Sheets. The abbreviation of enumerations must be specifically fashioned for each meta-model. The HTML syntax does not use the Math-ML standard. Math-ML is intended for the exchange and presentation of mathematical equations, while OCL is a computer language. The HTML syntax also does not use textual entities, although they could be mapped in an additional step.

For the mapping from concrete syntax, the text is first converted to HTML without any change, then, the regular expression are applied. In order to work on the encoded text, the regular expression also have to be encoded to use HTML conventions. For all textual matches, the result is identical. The only clashes with the HTML syntax arise for the equality ('<>') and collection operation tokens ('->'), which use the entities < and > instead.

For the mapping to concrete syntax, the process is reversed. First, the abbreviations are expanded to regular HTML, than HTML is converted back to concrete syntax.

3.3 Latex

Latex 2e and the ISO entity package with its transitive dependencies are the basis for the Latex mapping. All glyphs used in the abbreviations described above are

available in math mode. Thus, the OCL source code is translated to be included in a 'displaymath' environment. This allows simplified later editing of documents, because symbols do not have to be escaped. Further, it enables simple line numbering, using the 'equation' environment. Latex command syntax does not clash with OCL syntax, as the backslash character is not allowed in any identifiers; It is in the 'inhibitedChar' character set of the OCL 1.6 grammar[11]. Like in the case of Unicode, the translation from concrete syntax is straightforward. It can be reversed easily, after any additional latex formatting, like additional line breaks, has been removed.

3.4 Implementation

The translation is implemented as a Java class, which is based on the Java Regular expression facility, which provides an implementation based on the POSIX standard. The Microsoft .Net runtime also provides a regular expression engine. The translation tables described above are laid down in comma separated value files based on the ASCII character set. Unicode characters are abbreviated using the character escape mechanism. The Java class is provided with an input stream, encoding and direction, and provides an output stream. The class only applies the regular expression specified. If special processing steps are required, as is the case with HTML, the class is wrapped up with an additional layer, which provides the processing. In this implementation, a utility class from the Java version of the W3C HTMLTidy project is used to perform the encoding.

4 Examples and Savings

As stated in the introduction, the examples below were taken from the UML 1.4.2 standard. Obviously, the quality of the notation cannot be assessed by a simple count of original and reformatted glyphs. In our experience, page space required to fit all of the UML's WFRs was reduced about a quarter, while legibility seemed improved. This observation is obviously subjective. Especially the question, whether and when the advantage of brevity offsets the additional learning cost for the notation in novice users, probably cannot be decided without a larger work-efficiency study based on sound methods of organisational or didactic psychology.

On the examples below, it is worth noting, that in the layout of the UML standard, the WFRs take up 9 and 10 lines respectively. To allow a fairer comparison, the original source was reformatted not to break the page margin. Access space in the pretty printed version was used to convey logical structure of the statement.

4.5.3.10 Component [3] A Component may only have as residents DataTypes, Interfaces, Classes, Associations, Dependencies, Constraints, Signals, DataValues, and Objects.

$\circlearrowleft$.allResidentElements$\xrightarrow{\forall}$(re | re.κ?(DataType) $\vee$ re.κ?(Interface) $\vee$ re.κ?(Class) $\vee$ re.κ?(Association) $\vee$ re.κ?(Dependency) $\vee$ re.κ?(Constraint) $\vee$ re.κ?(Signal) $\vee$ re.κ?(DataValue) $\vee$ re.κ?(Object))

```
self.allResidentElements->forAll( re |
re.oclIsKindOf(DataType) or re.oclIsKindOf(Interface) or
re.oclIsKindOf(Class) or re.oclIsKindOf(Association) or
re.oclIsKindOf(Dependency) or re.oclIsKindOf(Constraint) or
re.oclIsKindOf(Signal) or re.oclIsKindOf(DataValue) or
re.oclIsKindOf(Object) )
```

4.10.3.4 Collaboration [1] All Classifiers and Associations of the ClassifierRoles and AssociationRoles in the Collaboration must be included in the namespace owning the Collaboration.

$\circlearrowleft$.allContents$\xrightarrow{\forall}$(e |
(e.κ?(ClassifierRole)$\Rightarrow$ $\circlearrowleft$.namespace.allContents$\xrightarrow{\in}$(e.τ!(ClassifierRole).base))$\wedge$
(e.κ? (AssociationRole)$\Rightarrow$ $\circlearrowleft$.namespace.allContents$\xrightarrow{\in}$(e.τ!(AssociationRole).base)))

```
self.allContents->forAll(e|(e.oclIsKindOf(ClassifierRole) implies
self.namespace.allContents->includes(e.oclAsType(ClassifierRole).base))
and (e.oclIsKindOf(AssociationRole) implies self.namespace.allContents
->includes (e.oclAsType(AssociationRole).base)))
```

5 Related Approaches

The Object-Z community uses a similar mechanism for dealing with different representations within the set of the Common Zed Tools (CZT)[10]. Here, a standard Unicode representation is used as the pivotal representation of the Zed and Object-Z languages to produce other renderings. However, that approach depends on a complete parse of the source code. Also, Unicode encoded Zed sources are quite rare and latex representation is not easily convertible into it. The B language also offers a similar facility within the jBTools suite[13], which allows conversion to HTML. However, in this suite, the full complexity of the B language is restricted on input from concrete syntax in order to allow conversion to an XML intermediate format known as B-XML. This is due to the fact that the aim of the suite is to provide further services beyond presentation, like type checkers and provers.

6 Summary and Outlook

In this paper we have shown how a flexible mechanism to abbreviate OCL concrete syntax of different versions can be defined, used to transfer code between main areas of use and flexibly implemented. Beyond this, the RE replacement approach could further be used to remedy some shortcomings within the standard and fix tool incompatibilities.

In this context it acts much like a macro-processor would, transforming an concrete syntax with abbreviations into an expanded form. Three examples for such a

scenario would be a templating mechanism for transitive closures, the avoidance of non-deterministic behaviour caused by the 'any' and 'asSequence' operations and fixing parser incompatibilities. We will focus on the last two examples here.

6.1 Non-deterministic Behaviour

As OCL is an expression language, and makes intensive use of iterators, optimisation options for its execution strongly depend on determinism of sub-expressions. Although expressive in theory[2], the 'any', 'asSequence' and 'iterate' operations violate execution determinism. 'any' yields a (potentially different) element of a collection on each execution. For this reason, the operation has been renamed 'random' in the above listing. With a macro mechanism as explained in the preceding section, it would be possible to replace the any operation with a deterministic variant specified (*def*ined) in OCL. As a result, there would not be ambiguities in the computational semantics. The 'asSequence' operation returns the content of a collection as a sequence. Order is non-deterministic. Any order-dependent operations based on the resulting list may hence yield different results. Here, the replacement mechanism could require the use of a sorted operation, whose comparison predicate definition has to be provided by the author. The predicate could be written to accept the most general type 'OclAny', and then list case choices for each class for which a comparison is implemented. Finally, the 'iterate' operation is defined on collections, which leads to the same problem as indicated for the 'asSequence' operation. With replacement, uses of iterate could generally be restricted to be prefixed with a cast to a list, using the safe version of 'asSequence' previously outlined.

6.2 Parser incompatibilities

OCLE of Babes-Bolyai University [4], the Dresden OCL Toolkit[9] and the Kent Modelling Framework [1] are three major tool suites for OCL. Except for the Dresden Toolkit, all of them use textual notation to interchange OCL and all use parsers with slightly different grammars. The differences between those notations are often minor. For example, OCLE uses an optional 'model' construct to denote the underlying model in a file, while the Dresden toolkit insists that a specification should start with a 'package' statement. KMF does not expect any structural parts, but works with OCL Expressions only. Such minor inconsistencies could be resolved with the same infrastructure used above to achieve the abbreviation markup.

References

1. David H. Akehurst and B. Bordbar. On querying UML data models with OCL. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 91–103. Springer, 2001.

2. Thomas Baar. Non-deterministic constructs in OCL – what does any() mean. In *Proc. 12th SDL Forum, Grimstad, Norway, June 2005*, volume 3530 of *LNCS*, pages 32–46. Springer, 2005.
3. S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford uiversity computingh Laboratory, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.
4. Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML models consistency using the OCL environment. *Electr. Notes Theor. Comput. Sci*, 102:99–110, 2004.
5. Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison Wesley Publisher, Reading, Mass., 1998.
6. Alexandre L. Correa and Cláudia Maria Lima Werner. Applying refactoring techniques to UML/OCL models. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *UML*, volume 3273 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2004.
7. Mark Davis. Unicode regular expression guidelines. Unicode Technical Report 18, The Unicode Consortium, P.O. Box 700519, San Jose, CA 95170-0519, USA, Phone: +1-408-777-5870, Fax: +1-408-777-5082, E-mail: `unicode-inc@unicode.org`, 1999.
8. Birgit Hofreiter, Christian Huemer, and Werner Winiwarter. OCL-constraints for UMM business collaborations. In Kurt Bauknecht, Martin Bichler, and Birgit Pröll, editors, *EC-Web*, volume 3182 of *Lecture Notes in Computer Science*, pages 174–185. Springer, 2004.
9. Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd International Conference on the Unified Modeling Language (UML)*, volume 1939 of *LNCS*, pages 278–293. Springer-Verlag, 2000.
10. Petra Malik and Mark Utting. CZT: A framework for Z tools. In *ZB*, pages 65–84, 2005.
11. Object Management Group (OMG). *Unified Modeling Language Specification, Version 1.4*, September 2001. `http://cgi.omg.org/docs/formal/01-09-67.pdf`.
12. Bernhard Rumpe. <<java>>OCL based on new presentation of the OCL-syntax. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*, pages 189–212. Springer, 2002.
13. Bruno Tatibouet. The jBTools b-suite for JEdit, 12 2003.
14. Rune Mathisen Vidar Bronken Gundersen. ISO character entities and their LATEX equivalents, 12 2001.

Author Index