Proceedings of the MoDELS'05 Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends

Thomas Baar (Ed.)

Technical Report No. LGL-REPORT-2005-001
September 2005

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Software Engineering Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland

Thomas Baar (Ed.)

Tool Support for OCL and Related Formalisms - Needs and Trends

Workshop co-located with MoDELS'05: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (formerly the UML Series of Conferences)

Montego Bay, Jamaica, October 4, 2005
Proceedings

Preface

This Technical Report comprises the final versions of all technical papers presented at the workshop *Tool Support for OCL and Related Formalisms - Needs and Trends* held in Montego Bay (Jamaica), October 4, 2005. The workshop was co-located with the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences) and continued a series of workshops focussing on OCL held at UML conferences in past years: 2000 in York, 2001 in Toronto, 2003 in San Francisco, and 2004 in Lisbon.

The frequency of workshops on OCL in the last years shows the genuine interest of both the research community as well as practitioners to discuss in detail the role OCL can play in precise modeling. While previous OCL workshops mainly aimed at analyzing the OCL itself and at clarifying the official language specification, the workshop this year concentrated on tool support and new application scenarios. The focus of the workshop is reflected by the titles of the three paper presentation sessions.

In session *Application of OCL*, the paper *On Squeezing M0, M1, M2, and M3 into a Single Object Diagram* by Gogolla, Favre, and Büttner applies OCL to formalize and clarify several metamodeling notions, which are currently used in a loose way by modelers. The paper *Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components* by Ackermann proposes a pattern-based technique to support the user in writing domain-specific OCL constraints.

In the second session *Tool Support for OCL*, three papers describing techniques for parsing and transforming OCL constraints were presented. The paper *Supporting OCL as part of a Family of Languages* by Akehurst, Howells, and McDonald-Maier analyzes a framework for defining the concrete syntax of a family of OCL-like languages. An OCL-like language shares large parts of its syntax with OCL but can also add new syntactic constructs or can redefine existing ones. The paper *Generation of an OCL 2.0 Parser* by Demuth, Hussmann, and Konermann identifies weaknesses of the current concrete syntax description for OCL 2.0 and makes suggestions how they can be mastered from the tool developer's point of view. The third paper of this session *Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java* by Dzidek, Briand, and Labiche reports on the application of aspect-oriented techniques to implement an OCL to Java translator.

The final session *History and Future of OCL* aimed at discussing trends in future uses of OCL. The papers also try to draw conclusions for the definition of OCL, including tool support. The paper *Proposals for a Widespread Use of OCL* by Chiorean, Bortes, and Corutiu formulates a proposal for defining different dialects of OCL that rely on a common core. Another topic is the support of such dialects by a single tool suite. In *OCL and Graph Transformations – A*

Symbiotic Alliance to Alleviate the Frame Problem by Baar, a combination of OCL with graph grammars is proposed. This combination is motivated by the frame problem, which has not been sufficiently addressed in OCL yet.

Finally, I would like to express my sincere gratitude to all members of the organizing committee for the lively discussion on the topic of this year's OCL workshop, for their dedication to writing reviews, and for useful suggestions for the final program. Last but not least, the authors of all submitted papers are gratefully thanked for having made this workshop possible. Might this workshop have given some inspiration to all workshop attendees. Might, furthermore, this inspiration culminate in interesting new papers on OCL, laying the foundations for scientific disputes at forthcoming workshops and conferences.

September 2005 Thomas Baar

Organization

Organizing Committee

Thomas Baar (Switzerland)
Dan Chiorean (Romania)
Alexandre Correa (Brazil)
Martin Gogolla (Germany)
Heinrich Hußmann (Germany)
Octavian Patrascoiu (UK)
Peter H. Schmitt (Germany)
Jos Warmer (The Netherlands)

Table of Contents

On Squeezing M0, M1, M2, and M3 into a Single Object Diagram

Martin Gogolla, Jean-Marie Favre, Fabian Büttner

University of Bremen (D), University of Grenoble (F), University of Bremen (D)

Abstract. We propose an approach for the integrated description of a metamodel and its formal relationship to its models and the model instantiations. The central idea is to use so-called layered graphs permitting to describe type graphs and instance graphs. A type graph can describe a collection of types and their relationships whereas an instance graph can represent instances belonging to the types and respecting the relationships required by the type graph. Type graphs and instance graphs are used iteratively, i.e., an instance graph on one layer can be regarded as a type graph of the next lower layer. Our approach models layered graphs with a UML class diagram, and operations and invariants are formally characterized with OCL and are validated with the USE tool. Metamodeling properties like strictness or well-typedness and features like potency can be formulated as OCL constraints and operations. We are providing easily understandable definitions for several metamodeling notions which are currently used in a loose way by modelers. Such properties and features can then be discussed on a rigorous, formal ground. This issue is also the main purpose of the paper, namely, to provide a basis for discussing metamodeling topics.

Keywords: System, Model, Metamodel, Meta-Metamodel, Class, Instance, InstanceOf, RepresentedBy, ConformsTo, Well-Typedness, Strictness, Potency, Layered Graph.

1 Motivation

Recent research activities and results in software engineering indicate that metamodeling is becoming more and more important [Sei03,Tho04,Bez05]. There are a lot of discussions about properties and notions of metamodels like the strictness of a metamodel or the potency of elements in it [AKHS03,AK03]. There are special sessions at scientific events on metamodeling. Standardized (e.g., by the OMG) and scientific, alternative metamodels have been developed for important languages like UML [ESW$^+$05], MOF, OCL [WK02,RG99], and CWM, to name only a few. A book on metamodeling is currently under development [CESW04]. Metamodeling is important within the Model Driven Architecture [Fra03,KWB03,MSUW04], and metamodeling is beginning to be broadened to megamodeling [Fav05b,Fav05a].

However, notions within the metamodeling area are often loose due to a lack of formalization. This has been recently referred to as the *meta-muddle*. Let us mention some examples. (A) A recent nice paper [Bez05] distinguishes between *System*, *Model*, *Metamodel*, and *Meta-Metamodel* whereas the conventional OMG approach uses the notion *User Objects* (*User Data*) instead of *System*. (B) In the same paper the author calls something a metamodel what would be called a model in the OMG terminology. (C) There are continuing discussions on whether the metamodels for UML 1 and UML 2 are strict or not.

The aim of this paper is to present a framework for discussing such notions and properties of metamodels by formalizing them. We will use a graph-based approach. Within metamodels one usually has different layers of abstractions, and each layer is more or less formally described. However, the relationship between the different layers is usually not explicitly discussed and described only implictly. Our approach allows to formally describe the different layers as well as the connection between the layers in an abstract form. Each layer will build a graph with nodes and edges, and also the connections between the layers will be formally described by edges. Thus we will obtain a comprehensive single graph covering the metamodel layers and their connections.

The structure of the rest of the paper is as follows. Section 2 will introduce the basic idea by means of a simple example. Section 3 formally shows our approach with its underlying class diagram including invariants and operations. Section 4 discusses further examples. The paper is finished with a concluding section which also contains open questions.

2 Describing Layered Graphs as Object Diagrams

Our layered graphs allow to organize complex metamodel structures into several abstraction layers. As an example, consider the graph in the left part of Fig. 1 possessing three layers: On the bottom layer one identifies nodes like ada and edges like ada_ibm; in the middle layer there are the nodes Person and Company and the edge Job; the top layer consists of the node Thing and the edge Connection. We discuss features of layered graphs by considering, for the example, the three layers one after the other, starting with the middle layer.

Middle layer: The middle layer can be thought of as representing a class diagram with two classes and one association.

Bottom layer: The bottom layer can be regarded as an object diagram with respect to the middle layer. The nodes and edges on this layer are all typed by dashed edges going to the middle layer. For example, the node ada is typed with a dashed edge going to node Person and the edge ada_ibm is typed with a dashed edge to the edge Job. Both typing elements, i.e., Person and Job, belong to the next higher layer.

Top layer: The top layer can be thought of as showing a class diagram such that the middle layer diagram becomes an object diagram for this top layer

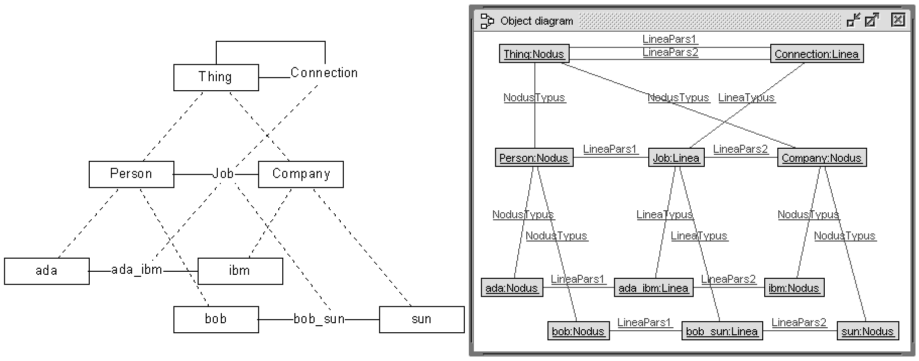


Fig. 1. Job Example as a Layered Graph and as an Object Diagram

diagram. The nodes Person and Company are typed as being nodes of type Thing and Job is an edge of type Connection.

The aim of the paper is to propose a general model for graphs like the one depicted in the left part of Fig. 1 but where not only three layers but an arbitrary number of layers can be captured. Each layer may have (solid) nodes and (solid) edges which can be typed by (dashed) edges to the next higher layer. We call such a graph consisting of several layers and (dashed) typing edges between the layers a *layered graph*.

The USE [RG01] screenshot in right part of Fig. 1 shows how the graph from the left part is represented as a UML object diagram. We will first explain the basic structure of the object diagram and introduce the respective class diagram later. One basic observation is that the nodes and the solid (non-dashed), named edges from the left part are represented as objects, but the dashed, unnamed edges are represented as links. The objects belong to (A) the class Nodus which realizes the nodes from the left part or (B) the class Linea which realizes solid edges from the left part. In order to have new, neutral names we have chosen the respective latin words as class names (see [Fav05b,Fav05a] for a discussion of the importance of distinguishing metamodel levels). Objects can be typed by Typus links. Nodus objects are typed by NodusTypus links, and Linea objects by LineaTypus links. The Typus links have been shown in the left part of Fig. 1 by dashed edges. Linea objects indicate their participating Nodus objects with links labelled LineaPars1 and LineaPars2. The latin word pars means part.

A layered graph does not induce a unique object diagram: For example, instead of having the link labelled LineaPars1 from Job to Person and the link labelled LineaPars2 from Job to Company, we could exchange 1 and 2 and have a link LineaPars2 from Job to Person and a link LineaPars1 from Job to Company.

3 Class Diagram, Operations, and Invariants

The class diagram in Fig. 2 shows our modeling for layered graphs by introducing the classes Nodus and Linea, the associations NodusTypus, LineaTypus, LineaPars1, and LineaPars2, and the names of the invariants.

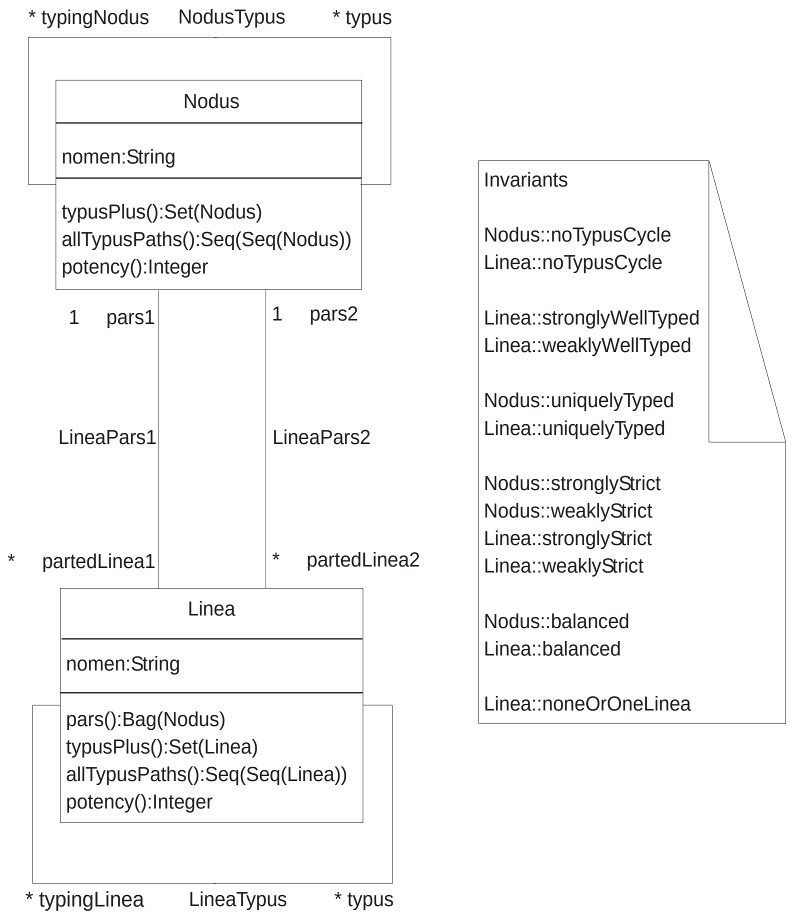


Fig. 2. UML Class Diagram for Nodus and Linea

– The class Nodus describes nodes. Nodus objects as well as Linea objects possess a string-valued name attribute nomen. The operation typusPlus is the transitive closure of the role name typus which will be explained below. The operation potency provides one definition for the potency of Nodus objects which basically indicates the layer to which the Nodus object belongs. The operation allTypusPaths delivers all paths without node repetitions consisting of Nodus objects starting in the respective Nodus object and using typing edges (dashed edges) only.

- The class Linea describes edges. Apart from having analogous operations mentioned already for the class Nodus, it possesses the operation pars yielding the bag of Nodus objects the respective Linea object is connected to.
- The association NodusTypus represents the typing of Nodus objects. Its role names are typus and typingNodus. typus yields the types of the current nodus object in the next higher layer (dashed edges upwards), whereas typingNodus yields the objects in the next lower layer which are typed through the current nodus object (dashed edges downwards). The association LineaTypus represents the typing of Linea objects. This association has analogous role names as NodusTypus. We emphasize that the multiplicities for NodusTypus and LineaTypus do not require a unique typing mechanism: Nodus and Linea objects can have multiple types.
- The associations LineaPars1 and LineaPars2 give the two Nodus objects the Linea object is connected to. These two Nodus objects have not to be distinct as, for example, the edge Connection in Fig. 1 shows. The choice between what is LineaPars1 and what is LineaPars2 is not important.

We now turn to the invariants. Not all these invariants are expected to be true in all system states which we discuss. The invariants are used to display whether certain properties hold in the current system state or not. Our USE tool allows with the invariant window to display such properties in a compact way, and therefore, we have formulated most properties as invariants. Alternatively, we could have formulated the invariants with observer operations, but then we could not have displayed the resuls in a compact way. However, the interesting discussion points will occur when certain invariants are not satisfied. Our introductory example however satisfies all invariants.

Most invariants are formulated for the class Nodus as well as for the class Linea. Therefore, we only show the invariants for class Nodus because the ones for class Linea are formulated analogously.[1]

-
  ```
  context self:Nodus inv noTypusCycle: -- also for Linea
    self.typusPlus()->excludes(self)
  ```
  Nodus::noTypusCycle requires that dashed edges between nodes do not include a cycle: The association NodusTypus constitutes a directed, acyclic graph (dag).
-
  ```
  context self:Linea inv stronglyWellTyped: -- only for Linea
    self.typus->notEmpty implies
      self.pars().typus=self.typus.pars()
  ```
  Linea::stronglyWellTyped demands that the types of the nodes of a (solid) edge are equal to the nodes of the types of the edge (types of the nodes vs. nodes of the types). In other words, for any (solid) edge, typing and building edge components are interchangeable.

[1] OCL invariants without explict variables possess an implicit variable `self` typed by the context class. We here prefer to name variables explicitly because we need a second context class variable for this constraint.

-
  ```
  context self:Linea inv weaklyWellTyped: -- only for Linea
    self.typus->notEmpty implies
      self.pars().typus->includesAll(self.typus.pars())
  ```
  Linea::weaklyWellTyped claims that the types of the nodes of a (solid) edge are a superset of the nodes of the types of the edge (again, types of the nodes vs. nodes of the types). In other words, for any (solid) edge, calculating first the type and then the edge components is compatible with calculating first the edge components and then the type, but not the other way round.
  The last two well-typedness properties can be formulated only for the class Linea.

-
  ```
  context self:Nodus inv uniquelyTyped: -- also for Linea
    self.typus->notEmpty implies self.typus->size=1
  ```
  Nodus::uniquelyTyped means that all nodes except the nodes in the top layer have exactly one type.

-
  ```
  context self:Nodus inv stronglyStrict: -- also for Linea
    self.typus->notEmpty implies
      self.typus->forAll(n|self.potency()+1=n.potency())
  ```
  Nodus::stronglyStrict demands that the potency of a node lies exactly under all the potencies of the types of the node being on the next higher layer.

-
  ```
  context self:Nodus inv weaklyStrict: -- also for Linea
    self.typus->notEmpty implies
      self.typus->forAll(n|self.potency()+1<=n.potency())
  ```
  Nodus::stronglyStrict requires that the potency of a node lies under, but not necessarily exactly under all the potencies of the types of the node being on the next higher layer.

-
  ```
  context self:Nodus inv balanced: -- also for Linea
    Nodus.allInstances->forAll(self2|
      self<>self2 and self.potency()=self2.potency() implies
      self.typingNodus->notEmpty=self2.typingNodus->notEmpty)
  ```
  Nodus::balanced states that all layers are balanced in the sense that two different nodes with the same potency also both possess typing nodes.

-
  ```
  context self:Linea inv noneOrOneLinea: -- only for Linea
    Linea.allInstances->forAll(self2|
      self<>self2 implies self.pars()<>self2.pars())
  ```
  Linea::noneOrOneLinea requires that between two nodes there can be at most one (solid) edge. This invariant could also be formulated in a restricted way for particular layers only.
- The overall aim of operation[2] potency shown in Fig. 3 is to return the layer number in which the respective Nodus object lies. The numbering starts with zero on the lowest layer. The formal definition of potency is rather complex, partly because potency should yield a result even in cases when the underlying NodusTypus structure is cyclic. The operation potency uses the helper operations max and allTypusPaths. allTypusPaths in turn needs the helper operation oneStep. allTypusPaths computes all paths consisting of nodes and (dashed) edges going upwards but a single node is allowed to occur only once. So, if there are cycles in the dashed edges between nodes,

[2] We employ the USE syntax for operation definitions.

i.e., invariant Nodus::noTypusCycle is not valid, allTypusPaths and with this potency will yield no usable result.

```
oneStep(aSeq:Sequence(Sequence(Nodus))):Sequence(Sequence(Nodus))=
  if aSeq->isEmpty or aSeq->isUndefined then
    oclEmpty(Sequence(Sequence(Nodus)))
  else
    aSeq->iterate(s:Sequence(Nodus);
      r1:Sequence(Sequence(Nodus))=oclEmpty(Sequence(Sequence(Nodus)))|
        s->last.typus->iterate(n:Nodus;r2:Sequence(Sequence(Nodus))=r1|
          if s->excludes(n)
            then r2->append(s->including(n)) else r2 endif))
  endif
allTypusPaths():Sequence(Sequence(Nodus))=
  Nodus.allInstances->iterate(n:Nodus;
    r:Sequence(Sequence(Nodus))=Sequence{Sequence{self}}|
    let new=oneStep(r)->reject(s|r->includes(s)) in r->union(new))
max():Integer=
  Nodus.allInstances->collect(n|n.allTypusPaths())->flatten->
    collect(size)->iterate(i:Integer;r:Integer=0 |
      if i>r then i else r endif)
potency():Integer=
  max()-self.allTypusPaths()->collect(p|p->size)->iterate(
    i:Integer;r:Integer=0 | if i>r then i else r endif)
```

Fig. 3. Operation potency

In order to give a simple example, how the operation potency works, we show its results for the introductory example:

```
                                     potency
ada,bob,ibm,sun,ada_ibm,bob_sun        0
Person,Company,Job                     1
Thing,Connection                       2
```

After having formulated these abstract properties let us turn to some more examples in order to see how the invariants behave in concrete situations.

4 Further Examples

The first example is about the poodle fido where poodle in turn is regarded as a breed. The situation is displayed in Fig. 4 as a layered graph and in Fig. 5 as a USE object diagram. The (solid) edges express an InstanceOf association. fido is typed as an Object and as a Poodle, Poddle is typed as a Breed, as an Object and as a Class, and Breed is typed as a Class. Thus the invariant Nodus::uniquelyTyped is invalid. This formally reveals that Poodle is a clabject. A clabject is a cross between class and object [AKHS03,AK03]. In formal terms this can be captured as Poodle.typus = Set{Breed,Class,Object}. The typing is also not strongly strict, i.e., the invariant Nodus::stronglyStrict is invalid, because Poodle.potency()=1 but, e.g., Object.potency()=3. Last, the Linea objects are not strongly well-typed because, e.g., Poodle_fido.pars().typus = Bag{Breed,Class,Object,Object,Poodle} but Poodle_fido.typus.pars() = Bag{Class,Object}. However, Linea objects are weakly well-typed, because the second mentioned bag is included in the first bag.

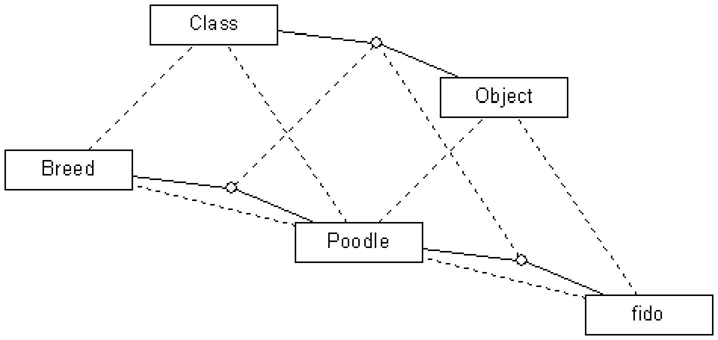
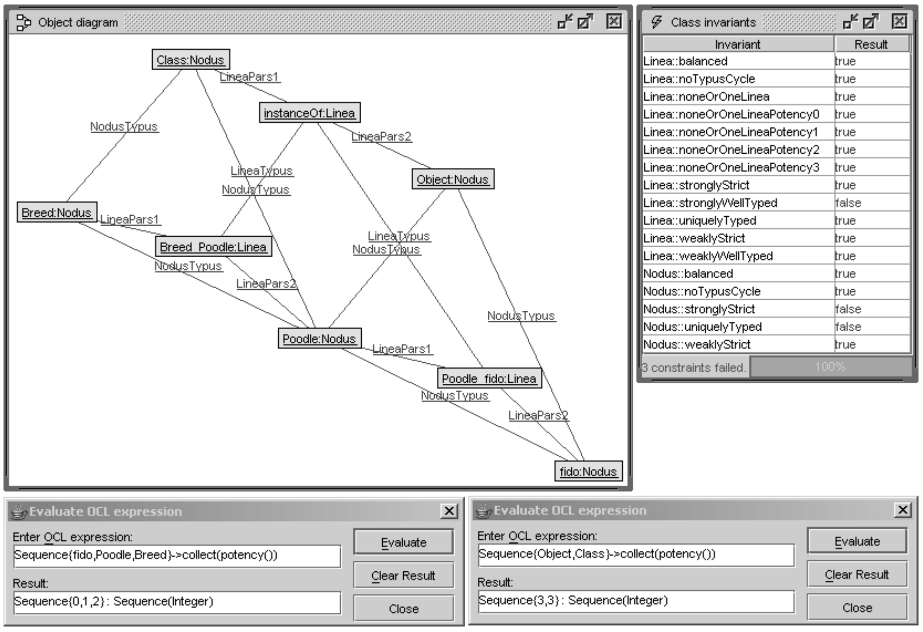


Fig. 4. The Fido Example as a Layered Graph



Invariant	Result
Linea::balanced	true
Linea::noTypusCycle	true
Linea::noneOrOneLinea	true
Linea::noneOrOneLineaPotency0	true
Linea::noneOrOneLineaPotency1	true
Linea::noneOrOneLineaPotency2	true
Linea::noneOrOneLineaPotency3	true
Linea::stronglyStrict	true
Linea::stronglyWellTyped	false
Linea::uniquelyTyped	true
Linea::weaklyStrict	true
Linea::weaklyWellTyped	true
Nodus::balanced	true
Nodus::noTypusCycle	true
Nodus::stronglyStrict	false
Nodus::uniquelyTyped	false
Nodus::weaklyStrict	true

3 constraints failed. 100%

Enter OCL expression:
Sequence{fido,Poodle,Breed}->collect(potency())
Result:
Sequence{0,1,2} : Sequence(Integer)

Enter OCL expression:
Sequence{Object,Class}->collect(potency())
Result:
Sequence{3,3} : Sequence(Integer)

Fig. 5. The Fido Example as a USE Object Diagram

The second example shows in Fig. 6 and Fig. 7 parts of a modeling for the Entity-Relationship model, the Relational data model and their translation. The left part expresses that the instance adaInstance is typed by PersonEntity and PersonEntity in turn is typed by Entity. The right part shows that the tuple adaTuple is typed by PersonRelSchema which in turn is typed by a Relational schema (RelSchema). Both elements, Entity and RelSchema, are typed as classes. The (solid) edges express a Reification connection between the elements. All invariants are satisfied and the potencies work in the hopefully expected way.

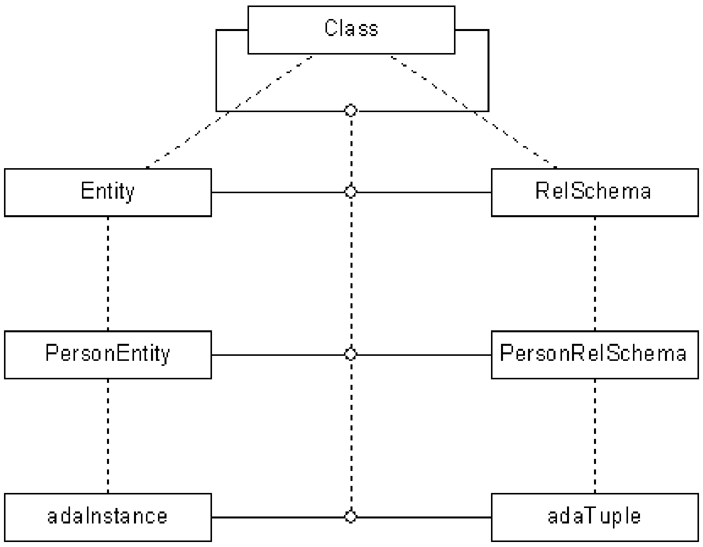
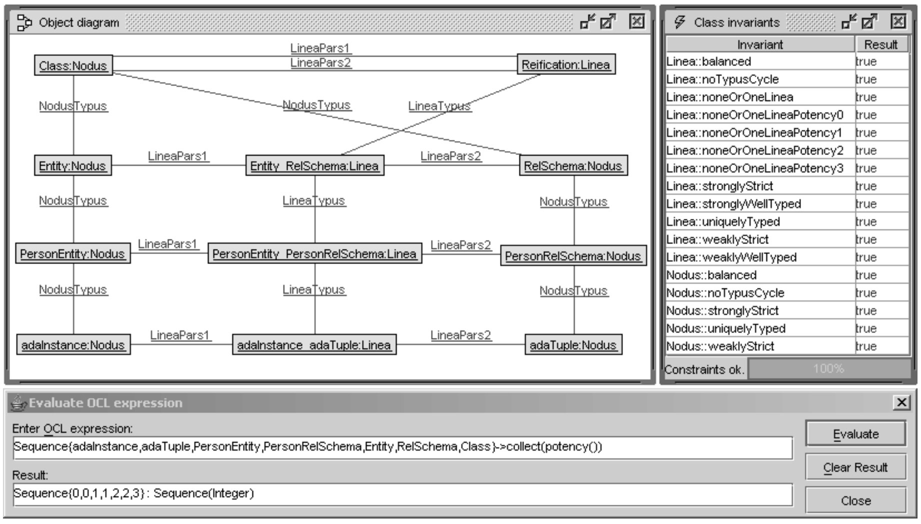


Fig. 6. The ER-RE Example as a Layered Graph



Invariant	Result
Linea::balanced	true
Linea::noTypusCycle	true
Linea::noneOrOneLinea	true
Linea::noneOrOneLineaPotency0	true
Linea::noneOrOneLineaPotency1	true
Linea::noneOrOneLineaPotency2	true
Linea::noneOrOneLineaPotency3	true
Linea::stronglyStrict	true
Linea::stronglyWellTyped	true
Linea::uniquelyTyped	true
Linea::weaklyStrict	true
Linea::weaklyWellTyped	true
Nodus::balanced	true
Nodus::noTypusCycle	true
Nodus::stronglyStrict	true
Nodus::uniquelyTyped	true
Nodus::weaklyStrict	true

Constraints ok. 100%

Enter OCL expression:

```
Sequence{adaInstance,adaTuple,PersonEntity,PersonRelSchema,Entity,RelSchema,Class}->collect(potency())
```

Result:

```
Sequence{0,0,1,1,2,2,3} : Sequence(Integer)
```

Fig. 7. The ER-RE Example as a USE Object Diagram

The third example in Fig. 8 and Fig. 9 shows a variation of the previous ER and Relational data model example where clabjects are used. As said already, interesting points come up where invariants are not satisfied. As in the first example of this section, Nodus::stronglyStrict and Nodus::uniquelyTyped fail. In this example, the Linea objects are not uniquely typed, but they are strongly well-typed. And the Linea object potencies are not strongly strict, because the (solid) edge potencies violate the requirements: PersonEntity_PersonRelSchema.potency()=1, but PersonEntity_PersonRelSchema.typus.potency()=Bag{2,3}.

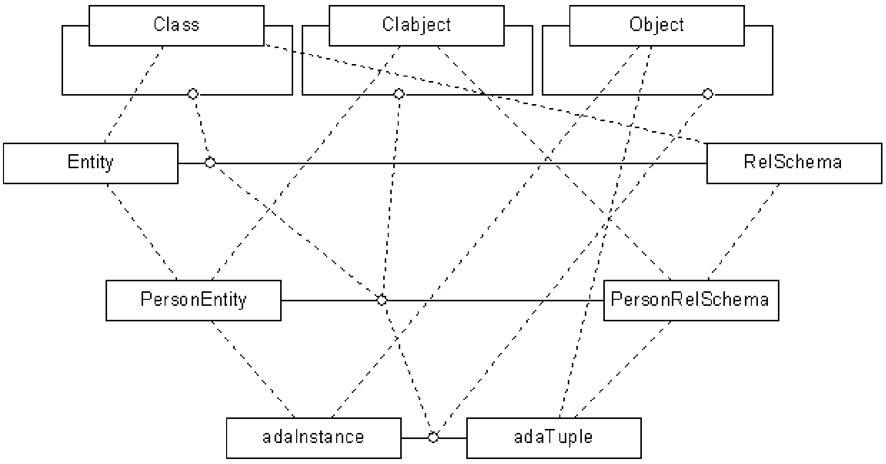


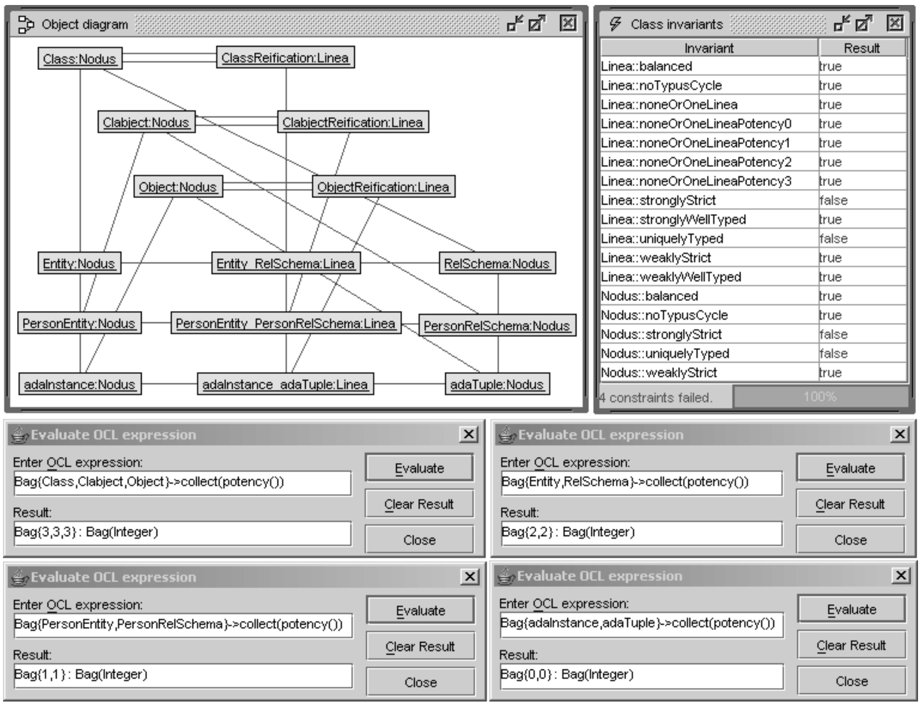
Fig. 8. Alternative ER-RE Example as a Layered Graph

5 Conclusion and Open Questions

This paper proposes to describe metamodeling notions and relationships with OCL invariants and operations. Such rigorous characterizations allow a precise and sharper discussion of such notions, the different metamodelling layers and their relationships. We have defined two versions of the notion *strict metamodeling* (weakly and strongly strict metamodeling), and we have formally defined the notion potency. An overview of our approach is given in Fig. 10. All four OMG metamodel layers are represented in a single object diagram which conforms to the Nodus-Linea class diagram. This is shown by the outer CD-OD bracket (CD-OD: Class Diagramm - Object Diagram). As indicated by the inner CD-OD brackets within this object diagram, further object and class diagrams with relationships between the layers are present. Note that in Fig. 10 the association Job and the link ada_ibm are displayed in the Nodus-Linea compliant style as nodes.

Future research includes the following questions.

– Fig. 10 assumes that the CD-OD brackets all obey the same rules or in other words that the class and object diagrams are UML respectively MOF diagrams. Are the inner CD-OD brackets and the outer CD-OD brackets really of the same kind?



Invariant	Result
Linea::balanced	true
Linea::noTypusCycle	true
Linea::noneOrOneLinea	true
Linea::noneOrOneLineaPotency0	true
Linea::noneOrOneLineaPotency1	true
Linea::noneOrOneLineaPotency2	true
Linea::noneOrOneLineaPotency3	true
Linea::stronglyStrict	false
Linea::stronglyWellTyped	true
Linea::uniquelyTyped	false
Linea::weaklyStrict	true
Linea::weaklyWellTyped	true
Nodus::balanced	true
Nodus::noTypusCycle	true
Nodus::stronglyStrict	false
Nodus::uniquelyTyped	false
Nodus::weaklyStrict	true

4 constraints failed. 100%

Evaluate OCL expression
Enter OCL expression:
```
Bag{Class,Clabject,Object}->collect(potency())
```
Result:
```
Bag{3,3,3} : Bag(Integer)
```

Evaluate OCL expression
Enter OCL expression:
```
Bag{Entity,RelSchema}->collect(potency())
```
Result:
```
Bag{2,2} : Bag(Integer)
```

Evaluate OCL expression
Enter OCL expression:
```
Bag{PersonEntity,PersonRelSchema}->collect(potency())
```
Result:
```
Bag{1,1} : Bag(Integer)
```

Evaluate OCL expression
Enter OCL expression:
```
Bag{adaInstance,adaTuple}->collect(potency())
```
Result:
```
Bag{0,0} : Bag(Integer)
```

Fig. 9. Alternative ER-RE Example as a USE Object Diagram

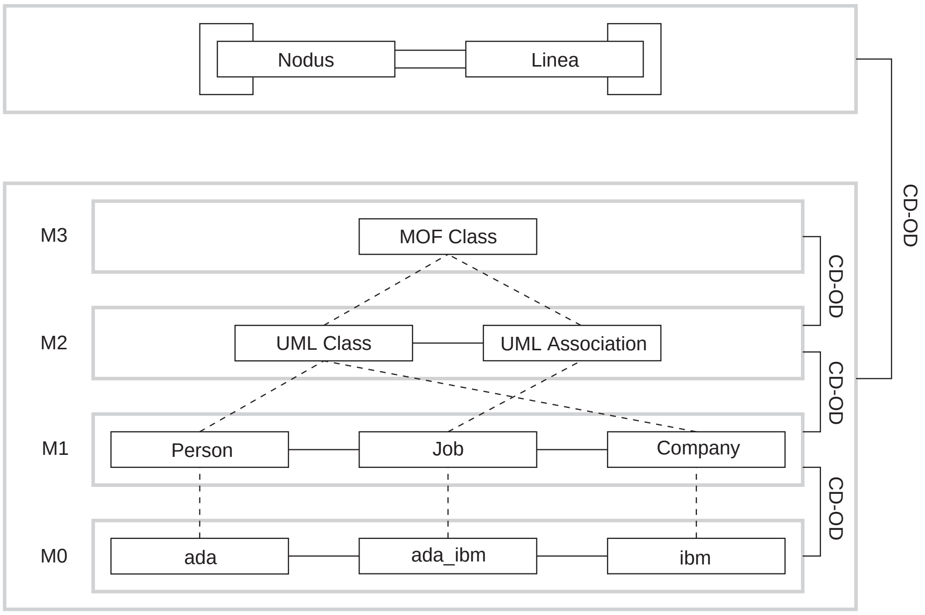


Fig. 10. Overview on the Approach

- The classes Nodus and Linea can be generalized to a common superclass GraphElement. This would make the model shorter but also even more abstract. Up to now our typing has to map nodes to nodes and edges to edges. A generalization class GraphElement could also allow, e.g., that nodes are type mapped to edges. Would this generalization condense the model or introduce more confusion?
- We have developed also an easier definition of the potency notion than the one we have shown. How does this easier definition relate to the more complex definition? Under which conditions do the two definitions coincide? How do these implicit or calculated potencies relate to an explicit assignment of potency in the class or object diagram?
- In our view a layer is collection of nodes and edges which have the same potency. And layers like M0, M1, M2, and M3 are then collections of objects and nodes with the respective potency. Is it possible to introduce particular constraints for particular layers? For example, naming conventions like *object names on the lowest layer (System) only have lower case letters*? Or, for example, uniqueness constraints for names within a particular namespace, like *Person names are unique*?
- In [Bez05] the view is taken that the *System* is *RepresentedBy* the *Model* and that the *Model* then *ConformsTo* the *Metamodel*. What are the consequences from the fact that both relationships *RepresentedBy* and *ConformsTo* are represented in our approach uniformly as dashed typing edges?
- Our approach also allows to discuss particular language feature of UML diagrams, e.g., ternary (and higher order) associations in class diagrams.

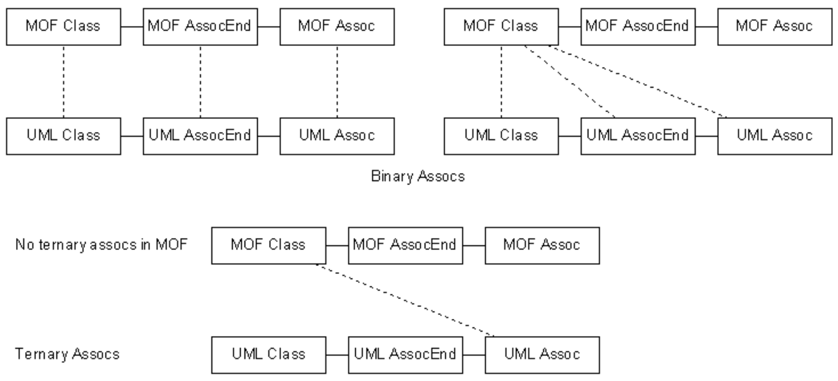


Fig. 11. UML in MOF - Ternary UML Assocs in MOF

What is the relationship between a UML Association and a MOF Association? In our approach, the question would be stated as: How is the typing of the Nodus object UML Association with respect to the next higher layer, i.e., with respect to the Nodus object MOF Association (as indicated in Fig. 11)? Goes the typing from UML Association to (A) MOF Class, (B) MOF Association, or (C) MOF Class and MOF Association?

- The general question behind this concrete question is: Where and how is the relationship between metamodeling layers expressed?
- Currently, our OCL tool USE (and also other tools) only allows to handle two layers: One class diagram and one object diagram layer. How can our approach help to develop *Meta-OCL tools* which support more layers where the middle layer (in a three layer setting) is at the same time an object diagram for the top layer and a class diagram for the bottom layer?

References

[AK03] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[AKHS03] C. Atkinson, T. Kühne, and B. Henderson-Sellers. Systematic Stereotype Usage. *Software and System Modeling*, 2(3):153–163, 2003.

[Bez05] J. Bezivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):171–188, 2005.

[CESW04] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied Metamodelling: A Foundation for Language Driven Development*. Xactium, 2004.

[ESW$^+$05] A. Evans, P. Sammut, J. S. Willans, A. Moore, and G. Maskeri. A Unified Superstructure for UML. *Journal of Object Technology*, 4(1):165–182, 2005.

[Fav05a] J.-M. Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[Fav05b] J.-M. Favre. Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In

J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[Fra03] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley and Sons, 2003.

[KWB03] A.G. Kleppe, J.B. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Pearson Education, 2003.

[MSUW04] S.J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled.* Addison Wesley, 2004.

[RG99] M. Richters and M. Gogolla. A Metamodel for OCL. In R. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, pages 156–171. Springer, Berlin, LNCS 1723, 1999.

[RG01] M. Richters and M. Gogolla. OCL - Syntax, Semantics and Tools. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[Sei03] E. Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[Tho04] D.A. Thomas. MDA: Revenge of the Modelers or UML Utopia? *IEEE Software*, 21(3):15–17, 2004.

[WK02] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley, 2nd Edition edition, 2002.

Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components

Jörg Ackermann

Chair of Business Informatics and Systems Engineering,
University of Augsburg, Universitätsstr. 16, 86135 Augsburg
joerg.ackermann@wiwi.uni-augsburg.de

Abstract. The Object Constraint Language (OCL) is often used for behavioral specification of software components. One current problem in specifying behavioral aspects comes from the fact that editing OCL constraints manually is time consuming and error-prone. To simplify constraint definition we propose to use specification patterns for which OCL constraints can be generated automatically. In this paper we outline this solution proposal and develop a way how to formally describe such specification patterns on which a library of reusable OCL specifications is based.

Keywords. Software Component Specification, OCL, Specification Patterns

1 Introduction

The Object Constraint Language (OCL) [20] has great relevance for component-based software engineering (CBSE): A crucial prerequisite for applying CBSE successfully is an appropriate and standardized specification of software components [27]. Behavioral aspects of components are often specified using OCL (see Sect. 2). From this results one of the current problems in component specifications: Editing OCL constraints manually is time consuming and error-prone (see Sect. 3).

To simplify constraint definition we propose to utilize specification patterns for which OCL constraints can be generated automatically (see Sect. 4). [4] identifies nine patterns that frequently occur in behavioral specifications of software components. In this paper we develop a solution how to formally describe specification patterns that enable a precise pattern specification and aid the implementation of constraint generators (Sect. 5). We conclude with discussion of related work (Sect. 6) and a summary (Sect. 7).

The main contributions of this paper are: the proposal to use specification patterns to simplify component specifications and the formal description of specification patterns by use of so called OCL pattern functions – together with the identified patterns we obtain a library of reusable OCL specifications. The results are not specific for software components and might therefore be interesting for any user of OCL constraints.

2 Specification of Software Components

The basic paradigm of component-based software engineering is to decouple the production of components (development for reuse) from the production of complete systems out of components (development by reuse). Applying CBSE promises (amongst others) a shorter time to market, increased adaptability and reduced development costs [8,25].

A critical success factor for CBSE is the appropriate and standardized specification of software components: the specification is prerequisite for a composition methodology and tool support [23] as well as for reuse of components by third parties [26]. With *specification* of a component we denote the complete, unequivocal and precise description of its external view - that is which services a component provides under which conditions [27].

Various authors addressed specifications for specific tasks of the development process as e.g. design and implementation [9,10], component adaptation [28] or component selection [15]. Approaches towards comprehensive specification of software components are few and include [7,23,27]. Objects to be specified are e.g. business terms, business tasks (domain-related perspective), interface signatures, behavior and coordination constraints (logical perspective) and non-functional attributes (physical perspective).

Behavioral specifications (which are topic of this paper) describe how the component behaves in general and in borderline cases. This is achieved by defining constraints (invariants, pre- and postconditions) based on the idea of designing applications by contract [18]. OCL is the de-facto standard technique to express such constraints – cf. e.g. [9,10,23,27].

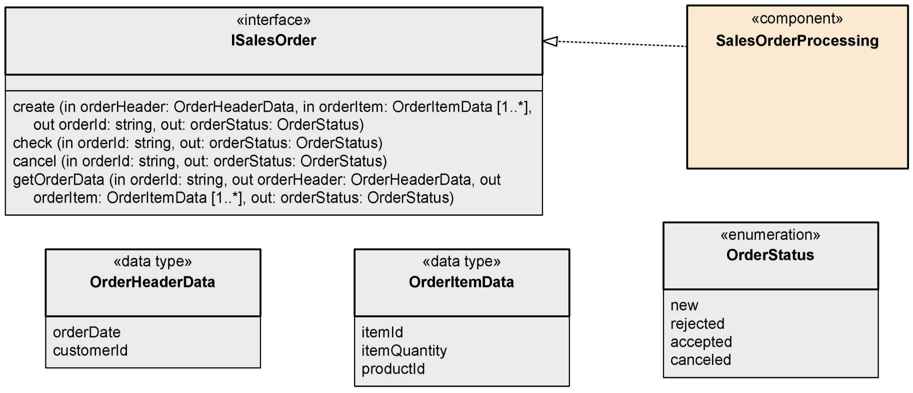

«interface»
ISalesOrder
create (in orderHeader: OrderHeaderData, in orderItem: OrderItemData [1..*], out orderId: string, out: orderStatus: OrderStatus)
check (in orderId: string, out: orderStatus: OrderStatus)
cancel (in orderId: string, out: orderStatus: OrderStatus)
getOrderData (in orderId: string, out orderHeader: OrderHeaderData, out orderItem: OrderItemData [1..*], out: orderStatus: OrderStatus)
«component»
SalesOrderProcessing
«data type»
OrderHeaderData
orderDate
customerId
«data type»
OrderItemData
itemId
itemQuantity
productId
«enumeration»
OrderStatus
new
rejected
accepted
canceled

Fig. 1. Interface specification of component *SalesOrderProcessing*

To illustrate how behavioral aspects of software components are specified we introduce a simplified exemplary component *SalesOrderProcessing*. The business task of the component is to manage sales orders. This component is used as example throughout the rest of the paper.

Fig. 1 shows the interface specification of *SalesOrderProcessing* using UML [21]. We see that the component offers the interface *ISalesOrder* with operations to create, check, cancel or retrieve specific sales orders. The data types needed are also defined in Fig. 1. Note that in practice the component could have additional operations and might offer additional order properties. For sake of simplicity we restricted ourselves to the simple form shown in Fig. 1 which will be sufficient as example for this paper.

To specify the information objects belonging to the component (on a logical level) one can use a specification data model which is realized as an UML type diagram and is part of the behavioral specification [3]. Fig. 2 displays such a model for the component *SalesOrderProcessing*. It shows that the component manages sales orders (with attributes id, date of order, status, customer id) and sales order items (with attributes id, quantity, product id) and that there is a one-to-many relationship between sales orders and sales order items.

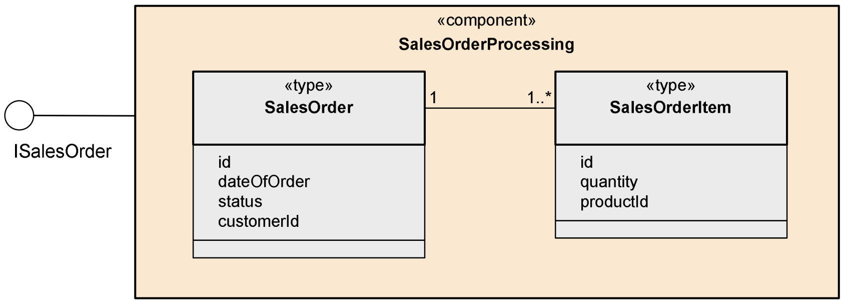


Fig. 2. Specification data model for component *SalesOrderProcessing*

Note that the interface *ISalesOrder* is connected to the component and not to one of the types in Fig. 2 – the types do not have operations at all. This is because the type diagram is only intended for specification purposes and shall not display internal realization details (black-box reuse). As a consequence the coupling between interface definition and type diagram is only loose. Of course one would expect that e.g. sales orders read by operation *ISalesOrder.getOrderData* correspond to the sales orders represented by type *SalesOrder*. This is, however, not automatically guaranteed and must be specified explicitly.

The behavioral specification of a component is based on its interface specification and on its specification data model and consists of OCL expressions that constrain the components operations – for an example see Fig. 3. The first constraint is an invariant for type *SalesOrder*: It guarantees that different sales orders always differ in the value of their id – that is the attribute *id* is a semantic key for sales orders. By defining an invariant this constraint needs only to be formulated once and does not need to be repeated in several pre-and postconditions. (Note that an invariant is supposed to hold for all component instances of a component installation within one system. But this distinction is not so important in our example because the component is by design a service-based component [16] which means that typically only one component instance is instantiated which serves all incoming requests.) The second constraint in Fig. 3 displays a precondition for operation *ISalesOrder.getOrderData*: The operation can only be called for a sales order that already exists in the component. (More precise: there must exist a sales order which id equals the value of the input parameter *orderId*. Note that the invariant guarantees that there is at most one such sales order).

```
context SalesOrder
 inv: SalesOrder.allInstances()->forAll(i1, i2 | i1 <> i2
                  implies i1.id <> i2.id)

context ISalesOrder::getOrderData(orderId: string, orderHeader:
OrderHeaderData, orderItem: OrderItemData, orderStatus: Order-
Status)
 pre: SalesOrder.allInstances()->exists(id = orderId)
```

Fig. 3. (Partial) Behavioral specification of component *SalesOrderProcessing*

3 Problems in Behavioral Specification of Components

Most component specification approaches recommend notations in formal languages since they promise a common understanding of specification results across different developers and companies. The use of formal methods, however, is not undisputed. Some authors argue that the required effort is too high and the intelligibility of the specification results is too low – for a discussion of advantages and liabilities of formal methods compare [14].

The disadvantages of earlier formal methods are reduced by UML OCL [20]: The notation of OCL has a simple structure and is oriented towards the syntax of object-oriented programming languages. Software developers can therefore handle OCL much easier than earlier formal methods that were based on set theory and predicate logic. This is one reason why OCL is recommended by many authors for the specification of software components.

Despite its advantages OCL can not solve all problems associated with the use of formal methods: One result of two case studies specifying business components [1,2] was the insight that editing OCL constraints manually is nevertheless time consuming and error-prone. Similar experiences were made by other authors that use OCL constraints in specifications (outside the component area), e.g. [13,17]. They conclude that it takes a considerable effort to master OCL and use it effectively.

It should be noted that behavioral aspects (where OCL is used) have a great importance for component specifications: In the specification of a rather simple component in case study [2], for example, the behavioral aspects filled 57 (of altogether 81) pages and required a tremendous amount of work. For component specifications to be practical it is therefore mandatory to simplify the authoring of OCL constraints.

4 Solution Proposal: Utilizing Specification Patterns

Solution strategies to simplify OCL specifications include better tool support (to reduce errors) and an automation of constraint editing (to reduce effort) – the latter can e.g. be based on use cases or on predefined specification patterns (compare Sect. 6).

To use specification patterns seems to be particularly promising for the specification of business components: When analyzing e.g. the case study [2] one finds that 70% of all OCL constraints in this study can be backtracked to few frequently occurring specification patterns. Based on this observation we analyzed a number of component specifications and literature about component specification and identified nine specification patterns that often occur [4]. These specification patterns are listed in Table 1. Although the nine patterns occurred most often in the investigated material there will be other useful patterns as well and the list might be extended in future.

Table 1. Behavioral specification patterns identified in [4]

Constraint type	Pattern name
Invariant	Semantic Key Attribute
Invariant	Invariant for an Attribute Value of a Class
Precondition	Constraint for a Input Parameter Value
Precondition	Constraint for the Value of an Input Parameter Field
Precondition	Instance of a Class Exists
Precondition	Instance of a Class does not Exist
Postcondition	Instance of a Class Created
Definition	Variable Definition for an Instance of a Class
Precondition	Constraint for an Instance Attribute for an Operation Call

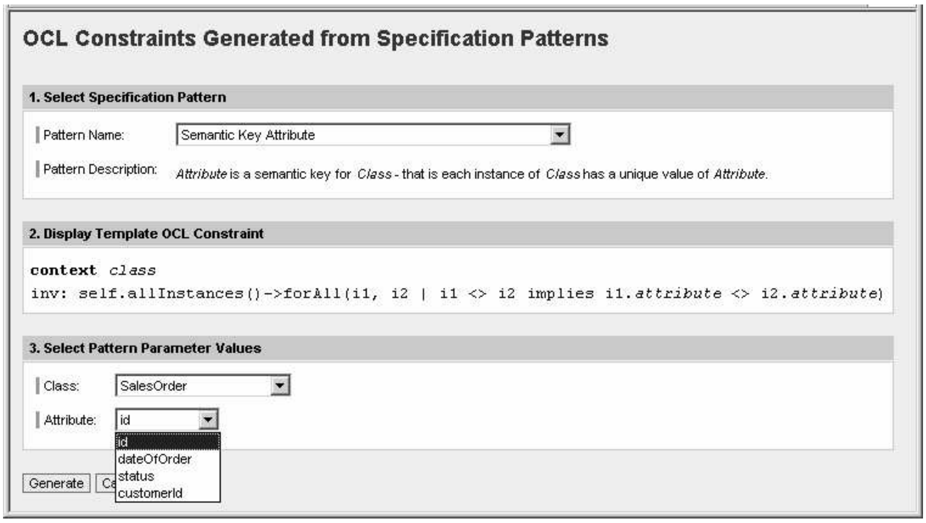
Under *(OCL) specification pattern* we understand an abstraction of OCL constraints that are similar in intention and structure but differ in the UML model elements used. Each pattern has one or more *pattern parameters* (typed by elements of the UML metamodel) that act as placeholder for the actual model elements. With *pattern instantiation* we denote a specific OCL constraint that results from binding the pattern parameters with actual UML model elements.

As an example let us consider the pattern "*Semantic Key Attribute*": It represents the situation that an attribute of a class (in the specification data model – cf. Fig. 2) plays the semantic role of a key – that is all instances of the class differ in their value of the key attribute. Pattern parameters are *class* and *attribute* and a pattern instantiation (for the class *SalesOrder* and attribute *id*) can be seen in the upper part of Fig. 3.

Table 2. Description scheme for pattern *Semantic Key Attribute* [4]

Characteristic	**Description**
Pattern name	Semantic Key Attribute
Pattern parameter	class: Class; attribute: Property
Restrictions	*attribute* is an attribute of class *class*
Constraint type	Invariant
Constraint context	*class*
Constraint body	`name(class).allInstances()->forAll(i1, i2 | i1 <> i2 implies i1.name(attribute) <> i2.name(attribute))`

Based on the ideas of [11] we developed a description scheme that details the properties of a specification pattern: pattern name, pattern parameters, restrictions for pattern use as well as type, context and body of the resulting constraint [4]. Note that the constraint body is a template showing text to be substituted in italic. The description scheme for the pattern *Semantic Key Attribute* is displayed in Table 2.


OCL Constraints Generated from Specification Patterns

1. Select Specification Pattern

Pattern Name: Semantic Key Attribute

Pattern Description: *Attribute* is a semantic key for *Class* - that is each instance of *Class* has a unique value of *Attribute*.

2. Display Template OCL Constraint

```
context class
inv: self.allInstances()->forAll(i1, i2 | i1 <> i2 implies i1.attribute <> i2.attribute)
```

3. Select Pattern Parameter Values

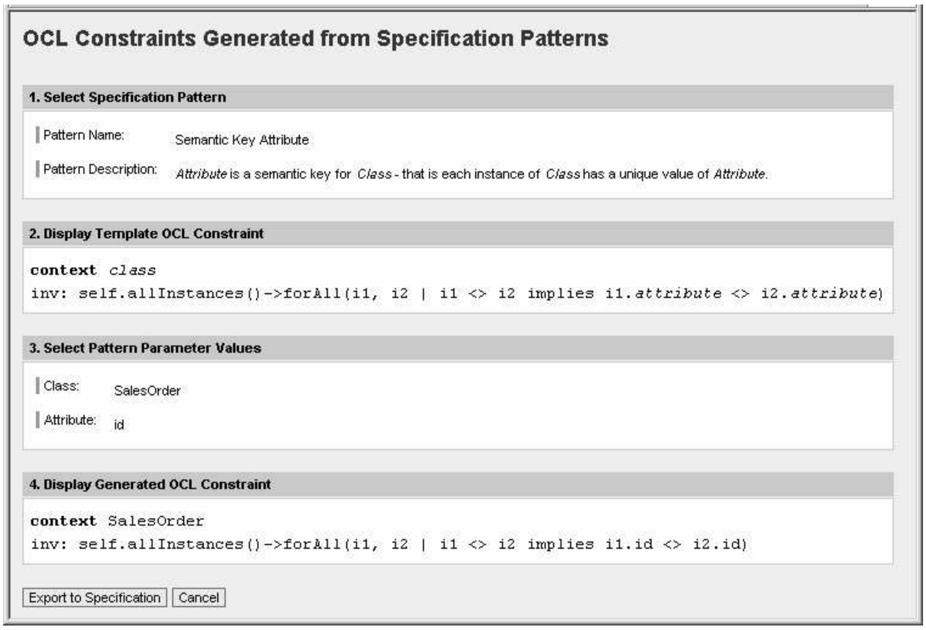
Class: SalesOrder

Attribute: id

Fig. 4. Selection screen for generating an OCL constraint

The following points connected with the exemplary pattern are worth mentioning: For sake of simplicity we presented the pattern with only one key attribute. In its regular version the pattern allows that the key is formed by one or more attributes of the class. (Note that this is the reason for not using the operator *isUnique* which would be rather constructed for more than one attribute.) One can also see that the patterns presented here are rather static – they allow for substituting UML model elements but do not allow for structural changes. For structural variations on the pattern (e.g.: the attribute *id* of class *SalesOrderItem* in Fig. 2 is only unique in the context of a specific instance of class *SalesOrder*) one has to define additional patterns.

We will now illustrate how such patterns can be exploited for specifications: Suppose the person who specifies our exemplary component is in the middle of the specification process and wants to formulate the invariant from Fig. 3. He checks the library of predefined specification patterns (which is part of his specification tool) and finds the pattern for a semantic key attribute (compare section 1 of Fig. 4). After selecting this pattern the tool will show him the pattern description and an associated template OCL constraint (showing the pattern parameters in italic). The user has to select model elements for the parameters (in section 3 of Fig. 4) – in our example the class *SalesOrder* and its attribute *id* are selected. Note that the tool can be built in such a way that it restricts the input to those model elements that are allowed for a pattern – in section 3 of Fig. 4 for instance you can see that the tool only offers the attributes of class *SalesOrder* for selection. After providing pattern and parameter values the user can start the generation. The tool checks the input for consistency and then generates the desired OCL constraint (compare section 4 of Fig. 5) which can be included into the component specification.



OCL Constraints Generated from Specification Patterns

1. Select Specification Pattern

Pattern Name: Semantic Key Attribute

Pattern Description: *Attribute* is a semantic key for *Class* - that is each instance of *Class* has a unique value of *Attribute*.

2. Display Template OCL Constraint

```
context class
inv: self.allInstances()->forAll(i1, i2 | i1 <> i2 implies i1.attribute <> i2.attribute)
```

3. Select Pattern Parameter Values

Class: SalesOrder

Attribute: id

4. Display Generated OCL Constraint

```
context SalesOrder
inv: self.allInstances()->forAll(i1, i2 | i1 <> i2 implies i1.id <> i2.id)
```

Fig. 5. Display of the generated OCL constraint

Following this approach has the following advantages: For the specification provider maintenance of specifications is simplified because it becomes faster, less error-prone and requires less expert OCL knowledge. For a specification user the understanding of specifications is simplified because generated constraints are uniform and are therefore easier recognizable. Moreover, if the patterns were standardized, it would be enough to specify a pattern and the parameter values (without the generated OCL text) which would make recognition even easier.

5 Technical Details of the Solution

To realize the solution outlined in Sect. 4 we need a way to formally describe the specification patterns. Such a formal pattern description is on one hand prerequisite for a tool builder to implement corresponding constraint generators – on the other hand it might also be interesting for a user creating specifications to check if a pattern meets his expectations (although one would not generally expect that a user has the knowledge to understand the formal pattern specifications). In this section we discuss how the specification patterns can be formalized and be described such that their intention, structure and application become unambiguous.

To do so we first show how such patterns can be formally described and applied (Sect. 5.1). After that we discuss the relationship of the solution to the UML metamodel (Sect. 5.2), argue why we have chosen it compared to other approaches (Sect. 5.3) and cover some implementation aspects (Sect. 5.4).

5.1 Defining OCL Pattern Functions for Specification Patterns

The basic idea how to formally describe the specification patterns is as follows: For each OCL specification pattern a specific function (called *OCL pattern function*) is defined. The pattern parameters are the input of the pattern function. Result of the pattern function is a generated OCL constraint which is returned and (if integrated with the specification tool) automatically added to the corresponding UML model element. The OCL pattern functions themselves are specified by OCL – from this specification one can determine the constraint properties (e.g. invariant) and its textual representation. All pattern functions are assigned as operations to a new class *OclPattern* which logically belongs to the layer of the UML metamodel (layer M2 in the four-layer metamodel hierarchy of UML [19] – compare also Sect. 5.2).

This approach will now be discussed in detail for the specification pattern "*Semantic Key Attribute*" (see Sect. 4). For this pattern we define the OCL pattern function *Create_Inv_SemanticKeyAttribute*. Input of the function are a class *cl* and an attribute *attr* which is the key attribute of *cl* – both understood as UML model elements. (To avoid naming conflicts with UML metamodel elements we did not use the pattern parameter names as displayed in the tool in Fig. 4 (like *class*) but more technical ones (as *cl*) as input parameters of the pattern functions.) Result is an UML model element of type *Constraint*. The complete specification of this pattern function is shown in Fig. 6.

```
context OclPattern::Create_Inv_SemanticKeyAttribute(cl: Class,
attr: Property): Constraint
(1) pre:  attr.class = cl

(2) post: result.oclIsNew
(3) post: result.namespace = result.context
(4) post: result.specification.isKindOf(OpaqueExpression)
(5) post: result.specification.language = 'OCL'

(6) post: result.stereotype.name = 'invariant'
(7) post: result.context = cl
(8) post: result.name = 'Semantic Key Attribute'
(9) post: result.specification.body =  OclPattern.Multiconcat
           (cl.name, '.allInstances()->forAll( i1, i2 | i1 <> i2
            implies i1.', attr.name, ' <> i2.', attr.name, ')')
```

Fig. 6. Specification of pattern function *OclPattern.Create_Inv_SemanticKeyAttribute*

The specification of each OCL pattern function consists of three parts:

- Preconditions specific for each pattern function (1)
- General postconditions (2)-(5)
- Postconditions specific for each pattern function (6)-(9).

The function specific preconditions describe which restrictions must be fulfilled when calling the pattern function. These preconditions must assure that the actual parameters conform to the specification pattern. For instance defines the signature of the pattern function in Fig. 6 only, that *cl* is any class and *attr* is any property. The precondition (1) demands additionally that *attr* is an attribute that belongs to class *cl.*

The general postconditions (2)-(5) are identical for all OCL pattern functions and represent in a way the main construction details. These postconditions (together with the functions signature) establish the following:

- The return of each pattern function is a UML model element of type *Constraint.*
- This constraint is added to the model (2) and is assigned to the model element which is the context of the constraint (3).
- The attribute *specification* of the constraint is of type *OpaqueExpression* (4) and is edited in the language OCL (5). (This is in conjunction with the newest version of OCL [20] from June 2005 – earlier there was an inconsistency in the OCL 2.0 specification. Compare Fig. 29 of [20].)

In difference to the general postconditions (2)-(5) the postconditions (6)-(9) vary between different pattern functions. The function specific postconditions establish the following:

- (6) describes of which constraint type (e.g. invariant, pre- or postcondition) the returned constraint is. The constraint of our example is an invariant.
- (7) defines the context of the constraint to be the class *cl.* The context of an invariant is always some class and the context of a pre- or postcondition is the classifier to which the operation belongs. Note that OCL imposes additional conditions depending on the constraint type. (An invariant, for instance, can only constrain one model element.) These additional constraints are part of the OCL specification [20, p. 176ff.] and will therefore not be repeated here.

- *Constraint* is a subtype of *NamedElement* and therefore has an attribute called *name* [21, p. 94]. This attribute is used in (8) where the constraint is assigned a name which is derived from the specification pattern (in our example the name *SemanticKeyAttribute*).
- The textual OCL representation of a constraint can be found in the attribute *body* of the property *specification* (which is of type *OpaqueExpression*) of the constraint. Postcondition (9) specifies this textual representation by combining fixed substrings (as *' <> i2.'*) with the name of model elements which were supplied as pattern parameter values (e.g. *cl.name*).

Note that standard OCL contains the function *concatenate* which allows concatenating two substrings. In postconditions like (9) of Fig. 6 it is necessary to concatenate many substrings. Technically one could do so by repeated application of OCL concatenate but the resulting expressions were hard to read. Instead we define a help function *OclPattern.Multiconcat*. Input of this function is a sequence of string arguments and its result is a string which is formed by repeated concatenation of the arguments (in the order given by the sequence).

```
constr := OclPattern.Create_Inv_SemanticKeyAttribute (SalesOr-
der, id)
```

Fig. 7. Call of pattern function *OclPattern.Create_Inv_SemanticKeyAttribute*

Fig. 7 shows how the pattern function *Create_Inv_SemanticKeyAttribute* is called in our example from Fig. 3: As values for the pattern parameters the class *SalesOrder* and the property *id* are used. The precondition is fulfilled because *id* is indeed an attribute of *SalesOrder*. The generated constraint *constr* is an invariant and its textual OCL representation is (as expected) the one shown as result in Fig. 5. (Due to missing UML syntax for operation calls we use in Fig. 7 a syntax that resembles the OCL syntax for operation calls.)

Other specification patterns can be described analogously. When defining OCL pattern functions one must be careful to select the correct UML metamodel elements for the pattern parameters (classes, properties (of classes), parameters, properties (of parameters) etc.) and to denote all relevant preconditions.

One aspect to be mentioned is that some specification patterns require pattern parameters with multiplicity higher than one. (In the regular version of the semantic key pattern there can be one or more attributes that form together the key of the class.) This can be solved by allowing input parameters of a pattern function to have multiplicity greater than one ([1..*]) and by employing the OCL operator *iterate* to construct the textual OCL specification in something like a loop.

5.2 Relationship with the UML Metamodel

The aim of this section is to discuss the relationship of the new class *OclPattern* with the UML language definition.

The UML metamodel is based on a four-layer metamodel hierarchy [19, p. 17ff.]: Layer M0 consist of the run time instances of model elements as e.g. the sales order with id ‘1234’. Layer M1 contains the actual user model in which e.g. the class *SalesOrder* is defined. Layer M2 defines the language UML itself and contains e.g. the model element *Class*. Note that layers M2 and M1 are the meta-layers for layers M1 and M0, respectively. Additionally there exists the layer M3 for the Meta Object Facility (MOF) which is an additional abstraction to define metamodels like UML.

For the constraint patterns we defined in Sect. 5.1 a new class *OclPattern*. To decide to which layer this class logically belongs we can analyze input and output of the pattern functions: Input of an OCL pattern function are elements of a UML model (like class *SalesOrder* or attribute *id* – on layer M1) that are typed by elements of the UML metamodel (like *Class* or *Property* – on layer M2). Analogously the output is always a constraint for a UML model element and is typed by the metamodel element *Constraint* (on layer M2). Consequently the pattern functions operate on layer M2 and therefore the new class *OclPattern* logically also belongs to layer M2.

On first glance it might seem desirable to integrate the class *OclPattern* into the UML metamodel (layer M2). The definition of UML, however, does not allow defining new elements in its metamodel. Adding the class *OclPattern* to layer M2 would effectively mean to define a new modeling language UML’ which consists of UML and one extra class – leaving standard UML yields to many disadvantages (potential compatibility and tool problems) and is not an adequate solution.

When looking more closely one finds that it is not necessary to integrate the class *OclPattern* that tightly into the UML metamodel because it does not change the language in the sense of introducing new model elements or changing dependencies.

As a conclusion it was decided: the class *OclPattern* will be denoted with the stereotype «oclHelper», operates on layer M2 but stands in parallel to the UML metamodel. The class needs only to be known to the specification tool implementing the constraint generators and is of no direct relevance for model users. The class might be integrated into the UML metamodel at a later time if the UML definition allows it. Note that on a related question OCL users asked to allow user defined OCL functions (Issue 6891 of OCL FTF) which was not realized in OCL 2.0.

5.3 Discussion of the Solution

In this section we will discuss the reasons why the approach presented in Sect. 5.1 was chosen and compare it with other solution approaches that seem (at least at first glance) possible.

By defining OCL pattern functions for the specification patterns it became possible to formally describe the patterns completely and quite elegantly: the pattern parameters can be found as function parameters and the function specification (which uses again OCL) describes the prerequisites to apply the pattern and the properties of the constraint to be generated. Moreover it is possible to actually specify that the constraint is added to the UML model element in consideration (assuming the pattern generator is integrated with the specification tool). One big advantage is that this approach only uses known specification techniques and does not require the invention of new ones. There is only one new class *OclPattern* that encapsulates the definition of all patterns.

An alternative approach would be to use a first-hand representation for the abstract constraints before parameter binding – [5] uses this approach and calls this representation *constraint schema*. The advantage is its explicit representation of the constraint schema. The disadvantage, however, is that constraint schemata are not defined in the UML metamodel – specifying them requires the invention of a special description technique (either outside UML or by introducing a new UML metamodel element). Therefore we decided against using this approach.

UML itself offers a mechanism called *Templates* that allows parameterizing model elements. The following approach seems to be promising and elegant: For each pattern one defines a template constraint which is parameterized by the pattern parameters – when applying the pattern these parameters are bound to the actual model elements. Unfortunately this solution is technically not possible because UML does not allow parameterizing Constraints (only Classifiers, Packages and Operations) [21, p. 600].

To use UML templates nevertheless one might think about parameterizing the context of a constraint (which is a classifier or an operation). But this approach is rather constructed and results in many disadvantages: For each *invariant* pattern used there needs to be a type in the specification data model and all business types using the pattern need to be bound to it. As a result the model would become overcrowded contradicting the clarity guideline from the guidelines of modeling [6]. (Similar problems occur with patterns of type pre- or postcondition where template operations need to be added to the interface model.)

5.4 Prototype Implementation

Constraint generators for specification patterns were implemented as a prototype (compare Fig. 4 and 5 in Sect. 4). The prototype enables to select a specification pattern and values for the pattern specific parameters. As far as possible pattern preconditions were considered when providing input for pattern parameters. All other preconditions must be checked after value selection. As a result the prototype generates the desired OCL constraint and displays it for the user. Planned for the future is an integration of constraint generators into a component specification tool – that would permit to automatically add the generated constraint to the correct model element of the UML model in work.

It shall be noted that the pattern parameters to be filled and the preconditions to be checked depend on the specification pattern – in the prototype these were hard coded. One could imagine something like a meta description that enables to (semi)automatically generate the constraint generator. The associated effort, however, seemed not appropriate for only nine specification patterns.

6 Related Work

Due to its importance component specifications are discussed by many authors (e.g. [9,10,23,27] – for an overview compare e.g. [23]). Most current specification approaches identify the need for behavioral specifications and propose to use pre- and postconditions based on OCL [20]. Problems related with using OCL were so far only reported in the case studies [1,2] and the author is not aware of any solution to this problem in the area of component specifications.

There are several publications outside the component area discussing the problems of editing OCL constraints manually [5,13,17]. There exist several approaches to simplify constraint writing: [13] develops an authoring tool that supports a developer with editing and synchronizing constraints in formal notation (OCL) and informal notation (natural language). [17] discusses an approach how to generate OCL expressions automatically. They constrain themselves, however, to the single use case of connecting two attributes within an UML model by an invariant. [12] discusses strategies to textually simplify OCL constraints that were generated by some algorithm. [24] develops an algorithm that allows in the analysis phase to transform use cases into class diagrams and OCL specifications. The author suggests that generation of OCL constraints might be possible but gives no details for it. [5] proposes a mechanism to connect design patterns with OCL constraint patterns which allows to instantiate OCL constraints automatically whenever a design pattern is instantiated.

The idea of [5] is very similar to our solution proposal. Its realization, however, can not be employed for specifying components: When instantiating a design pattern the mechanism of [5] always creates a new class (or several classes) into a class diagram together with the creation of the corresponding OCL constraints. This is not practical for component specification because the specification type diagram might already exist (e.g. derived from requirements definition [9]) or it might happen that (at specification time) several specification patterns need to be applied to a class in a combination not foreseen at pattern definition time. Moreover, the approach as presented in [5] is not integrated with the UML 2.0 metamodel, because its simple integration into the UML 1.3 metamodel (via properties of the type *oclType*) is not longer possible in UML 2.0.

7 Summary

The paper discussed one of the current problems in component specifications: editing OCL constraints manually is time consuming and error-prone. As solution we proposed to utilize specification patterns for which OCL constraints can be generated automatically. In this paper we developed a solution how to describe such specification patterns formally. As a result we achieved a precise pattern specification which aids e.g. the implementation of constraint generators. How such specification patterns can be utilized in the specification process was shown in a prototype implementation. Direction of future research include to gain more experience with the identified specification patterns (and extend the pattern library if needed) and to include the constraint generator functionality into the component specification tool [22] which is developed for the specification framework [27].

References

1. Ackermann, J.: Fallstudie zur Spezifikation von Fachkomponenten. In: Turowski, K. (ed.): 2. Workshop Modellierung und Spezifikation von Fachkomponenten. Bamberg (2001) 1-66 (In German)
2. Ackermann, J.: Zur Spezifikation der Parameter von Fachkomponenten. In: Turowski, K. (ed.): 5. Workshop Komponentenorientierte betriebliche Anwendungssysteme (WKBA 5). Augsburg (2003) 47-154 (In German)
3. Ackermann, J.: Spezifikation von Fachkomponenten mit der UML 2.0. In: Turowski, K. (ed.): 4. Workshop Modellierung und Spezifikation von Fachkomponenten. Bamberg (2003) 23-30 (In German)
4. Ackermann, J.: Frequently Occurring Patterns in Behavioral Specification of Software Components. In: *Turowski, K.; Zaha, J.M. (eds.):* Component-Oriented Enterprise Applications. Proceedings of the Conference on Component-Oriented Enterprise Applications (COEA 2005). Erfurt (2005) 41-56
5. Baar, T.; Hähnle, R.; Sattler, T.; Schmitt, P.H.: Entwurfgesteuerte Erzeugung von OCL-Constraints. In: Softwaretechnik-Trends 3 (2000) (In German)
6. Becker, J.; Rosemann, M.; von Uthmann, C.: Guidelines of Business Process Modeling. In: van der Aalst, W.; Desel, J.; Oberweis, A. (eds.): Business Process Management: Models, Techniques and Empirical Studies. Springer-Verlag. Berlin (2000) 30-49
7. Beugnard, A.; Jézéquel, J.-M.; Plouzeau, N.; Watkins, D.: Making Components Contract Aware. In: IEEE Computer 7 (1999) 38-44
8. Brown, A.W.: Large-Scale, Component-Based Development. Prentice Hall, Upper Saddle River (2000)
9. Cheesman, J.; Daniels, J.: UML Components. Addison-Wesley, Boston (2001)
10. D'Souza, D.F.; Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading (1998)
11. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
12. Giese, M.; Hähnle, R.; Larsson, D.: Rule-Based Simplification of OCL Constraints. In: Workshop on OCL and Model Driven Engineering at UML'2004. Lisbon (2004)
13. Hähnle, R.; Johannisson, K.; Ranta, A.: An Authoring Tool for Informal and Formal Requirements Specifications. In: Kutsche, R.-D.; Weber, H. (eds.): Fundamental Approaches to Software Engineering, 5th International Conference FASE. Grenoble (2002) 233-248
14. Hall, A.: Seven Myths of Formal Methods. In: IEEE Software 5 (1990) 11-19
15. Hemer, D.; Lindsay, P.: Specification-based retrieval strategies for module reuse. In: Grant, D.; Sterling, L. (eds.): Proceedings 2001 Australian Software Engineering Conference. IEEE Computer Society. Canberra (2001) 235-243
16. Herzum, P.; Sims, O.: Business Component Factory. Wiley Computer Publishing, New York (2000)
17. Ledru, Y.; Dupuy-Chessa, S.; Fadil, H.: Towards Computer-aided Design of OCL Constraints. In: Grundspenkis, J.; Kirikova, M. (eds.): CAiSE Workshops 2004, Vol. 1. Riga (2004) 329-338
18. Meyer, B.: Applying "Design by Contract". In: IEEE Computer 10 (1992) 40-51
19. OMG (ed.): Unified Modeling Language: UML 2.0 Infrastructure Specification. Finalized Convenience Document, 2004-10-16 URL: http://www.omg.org/technology/documents, Date of Call: 2005-09-09 (2004)
20. OMG (ed.): Unified Modeling Language: UML 2.0 OCL Specification, 2005-06-06. URL: http://www.omg.org/technology/documents, Date of Call: 2005-09-09 (2005)

21. OMG (ed.): Unified Modeling Language: UML 2.0 Superstructure Specification. Formal version, 2005-07-04. URL: http://www.omg.org/technology/documents, Date of Call: 2005-09-09 (2005)
22. Overhage, S.: Komponentenkataloge auf Basis eines einheitlichen Spezifikationsrahmens - ein Implementierungsbericht. In: Turowski, K. (ed.): Tagungsband des 3. Workshop Modellierung und Spezifikation von Fachkomponenten. Nürnberg (2002) 1-16 (In German)
23. Overhage, S.: UnSCom: A Standardized Framework for the Specification of Software Components. In: Weske, M.; Liggesmeyer, P. (eds.): Object-Oriented and Internet-Based Technologies, Proceedings of the 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (NODe 2004). Erfurt (2004)
24. Roussev, B.: Generating OCL specifications and class diagrams from use cases: A newtonian approach. In: Proceedings of 36th Annual Hawaii International Conference on System Sciences (HICSS'03). Big Island (2003)
25. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 2. ed. Addison-Wesley, Harlow (1998)
26. Turowski, K.: Spezifikation und Standardisierung von Fachkomponenten. In: Wirtschaftsinformatik 3 (2001) 269-281 (In German)
27. Turowski, K. (ed.): Standardized Specification of Business Components: Memorandum of the working group 5.10.3 Component Oriented Business Application System, February 2002. University of Augsburg, Augsburg (2002) URL: http://www.fachkomponenten.de. Date of Call: 2005-09-09
28. Yellin, D.; Strom, R.: Protocol Specifications and Component Adaptors. In: ACM Transactions on Programming Languages and Systems 19 (1997) 292–333

Supporting OCL as part of a Family of Languages

David H. Akehurst, Gareth Howells, Klaus D. McDonald-Maier

University of Kent at Canterbury
D.H.Akehurst@kent.ac.uk

Abstract. With the continued interest in Model Driven techniques for software development more and more uses are found for query or expression languages that navigate and manipulate object-oriented models. The Object Constraint Language is one of the most frequently used languages; however, its original intended use as a constraint expression language has been succeeded by its frequently proposed use as a basis for a more general model query language, model transformation language and potential action language. We see a future where OCL forms a basis for a family of languages related in particular to Model Driven Development techniques; as a consequence we require an appropriate tool suite to aid in the development of such language families. This paper proposes some important aspects of such a tool suit.

1 Introduction

The Object Constraint Language (OCL) is often the language of choice for adding precision to models or as a core component in model based expression languages. The Object Management group is currently going though the adoption process for version 2.0 of the OCL standard. However, there have been a number of problems with the specification as documented in papers such as [6, 7].

Despite problems with the standard the last few of years have seen the development of a number of OCL tools and modelling tools that support OCL; for example, the Dresden OCL tool kit [9], Octopus [13], Poseidon [10] and our own works [2, 3, 7] just to name a few.

In addition there have also been a number of papers suggesting extensions to the standard, e.g. adding relations [1], or using the OCL expression language as a basis for other languages such as an action language [11] or a model transformation language [4, 8, 15].

This use and reuse of OCL as a core expression language built on to provide other associated languages distinctly implies an ‘OCL Family’ of languages. Standard support for constructing tools for text based languages such as OCL do not generally support the notion of a language family, and few provide mechanisms for language reuse.

When constructing a family of languages, it is desirable if these parts can be reused, rather than redefined for each member of the family. This paper proposes a Framework for supporting such families of languages as plug-ins to the Eclipse [12] integrated development environment; focusing in particular on the specification of grammars to support reuse of the scanner and parser aspects of a language processor.

Parser Generators have been around for a long time; however there have been few advances towards enabling reuse of generated parsers or the grammar specifications used as input, other than basic cut and paste from one specification to another. This is primarily due to technical aspects of the algorithms used to implement efficient parsers. The Antlr [14] parser generator is one of the few that includes a grammar inheritance mechanism and we use something similar.

In this paper we define a family of OCL like languages using an EBNF based grammar specification language. The parsers for the languages have been implemented, based on the given specifications, and tested in an Eclipse based framework for enabling us to plug-in language processors.

2 An Eclipse Language Framework

To facilitate a mechanism to plug-in families of languages we define the concepts of *LanguageProcessor* and *Expression* as indicated in the class diagram of Figure 1. To access the range of potential languages available we also require a language Registry.

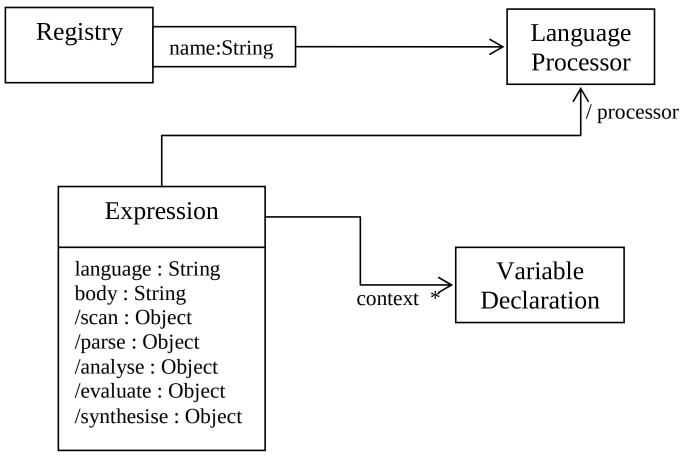


Figure 1 Language Framework

A *LanguageProcessor* is written and plugged-in in order to provide support for a particular language. An *Expression* can defined for any language, if a processor for that language has been provided, then it is used to provide scan, parse, analyse, evaluate and synthesis capabilities to the expression; or a subset of those capabilities. When the plug-in description for a language processor is specified, we can define which capabilities are supported by the processor.

We use this framework to plug-in our OCL family of languages.

3 A Family of OCL based Grammars

The following subsections illustrate a family of grammars using an object-oriented style wrapper for EBNF. The grammars specify the concrete syntax for a family of languages suitable for expressing queries, constraints, actions and transformations within an object-oriented and Model Driven Development based context. The grammars are based on the original OCL syntax although a few alterations have been made.

The grammar specifications are split into four packages:

1. oel – the base object expression language
2. ocl – an object constraint language
3. oql – an object query language
4. oal – an object action language

The overall relationship between the grammars is shown in Figure 2.

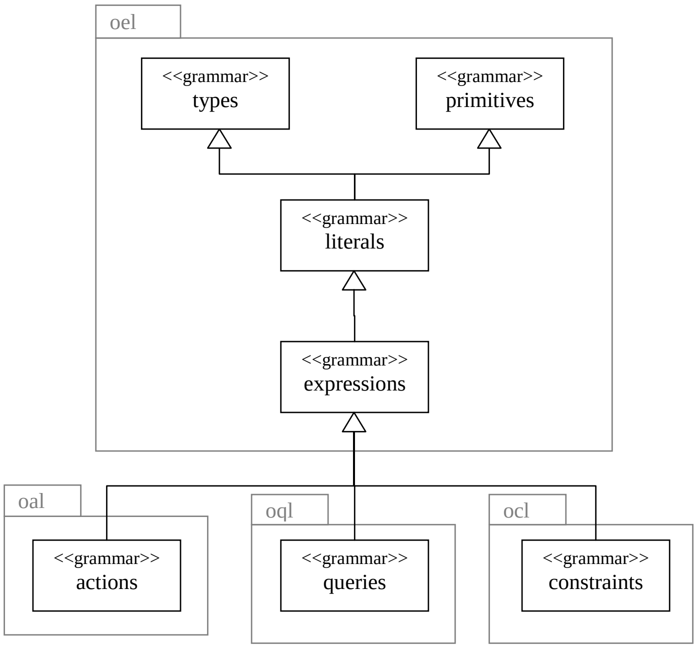


Figure 2 A Family of OCL like Languages

3.1 Object Expression Language (OEL)

The first, object expression language, we sub-divide into four related grammars, as this eases the specification and understanding of the language. The most basic aspect of *oel* is the specification of primitive values; the grammar for specifying these values is shown in Table 1.

```
package oel;

grammar primitives {
  primitive = BOOLEAN | STRING | INTEGER | REAL;
  BOOLEAN = 'true' | 'false' ;
  STRING = "\x27[^\x27]*\x27";
  INTEGER = "[0-9]+" ;
  REAL = "[0-9]+[.][0-9]+" ;
}
```

Table 1 Object Expression Language Primitives Grammar

In addition to primitive values we would also like to be able to refer to various types of object (classes or classifiers from a model) either directly or as collections or tuples. Table 2 shows a grammar for defining types; this grammar could also be used to define the valid type expressions used within graphical languages such as class diagrams.

```
package oel;

grammar types {
  pathName = NAME ( '::' NAME )* ;
  type = pathName | collectionType | tupleType ;
  collectionType = 'Set' '(' type ')'
                 | 'OrderedSet' '(' type ')'
                 | 'Sequence' '(' type ')'
                 | 'Bag' '(' type ')'
                 ;
  tupleType = 'Tuple' '(' variableDeclarationList ')' ;
  variableDeclarationList = variableDeclaration
                            (',' variableDeclaration)*;
  variableDeclaration = NAME ':' type ;
  NAME = "[a-zA-Z_][a-zA-Z_0-9]*" ;
}
```

Table 2 Object Expression Language Types Grammar

The next stage in the definition of the syntax of the object expression language is to providing syntax for defining literal values. The literals grammar is shown in Table 3; it extends (resues) both the *types* and *primitives* grammars, i.e. the production rules defined in *types* and *primitives* are semantically included in the definition of the literals grammar.

Note that there is an apparently redundant production rule *expression* used in the rules for *variableDefinition* and *collectionLiteralPart*, this production will be overridden in a sub grammar to facilitate more complex expressions to be used in the definition of variables and collection literal parts.

```
package oel;

grammar literals extends primitives, types {
  expression = literal;
  literal    = primitive
             | collection
             | tuple
             | type
             ;
  collection = 'Set' '{' [collectionLiteralPartList] '}'
             | 'OrderedSet' '{' [collectionLiteralPartList] '}'
             | 'Sequence' '{' [collectionLiteralPartList] '}'
             | 'Bag' '{' [collectionLiteralPartList] '}'
             ;
  collectionLiteralPartList = collectionLiteralPart
                              (',' collectionLiteralPart)* ;
  collectionLiteralPart = expression | range ;
  range = expression '..' expression ;
  variableDefinitionList = variableDefinition
                           (',' variableDefinition)*;
  variableDefinition = NAME [':' type] '=' expression ;
  tuple = 'Tuple' '{' variableDefinitionList '}' ;
}
```

Table 3 Object Expression Language Literals Grammar

The final part of the object expression language grammar is the definition of syntax for writing expressions; Table 4 shows the relevant grammar. There are a number of aspects to note in this definition. Firstly we redefine the rule for *type*, as there is no syntactic mechanism to distinguish a single name as a type or variable reference, the distinction must be made during syntactic analysis. We also introduce an additional bit of syntax into the standard for EBNF, which has no mechanism for specifying precedence; we use the '<' symbol to indicate a precedence amongst a number of choices (we do not show here a mechanism for indicating operator associativity).

Finally, we show in this syntax and extension to the standard OCL notion of an *if…then…else…endif* expression. The standard OCL semantics actually have a tertiary logic based on *true*, *false* and *undefined*, we feel that it would be useful to reflect this in the associated *if* construct.

```
package oel;

grammar expressions extends literals {
  type = collectionType | tupleType ;
  expression = NAME
             < literal
             < '(' expression ')';
             < navigationExpression
             < unaryCall
             < binaryInfixCall
             < ifExp
             < letExp
             ;

  navigationExpression = iterateCall
                       | iteratorCall
                       | collectionOperationCall
                       | operationCall
                       | qualification
                       | propertyCall
                       ;
  iterateCall = expression '->' 'iterate' '('
                variableDeclaration ';'
                variableDefinition '|' expression ')' ;
  iteratorCall = expression '->' NAME '(' declaratorList '|'
                expression ')' ;
  declaratorList = declarator (',' declarator)* ;
  declarator = NAME | variableDeclaration ;
  collectionOperationCall = expression '->' NAME '('
                           [argumentList] ')' ;
  operationCall = [expression '.'] NAME '(' [argumentList] ')' ;
  qualification = [expression '.'] NAME '[' argumentList ']' ;
  propertyCall = expression '.' NAME ;
  letExp = 'let' variableDefinitionList 'in' expression ;
  ifExp = 'if' expression 'then' expression
          'else' expression 'undefined' expression 'endif' ;
  unaryCall = '-' expression | 'not' expression ;
  binaryInfixCall = expression '/' expression
                  < expression '*' expression
                  < expression '+' expression
                  < expression '-' expression
                  < expression ('div' | 'mod') expression
                  < expression ('and' | 'or'
                    | 'xor' | 'implies') expression
                  < expression ('>' | '<' | '>=' | '<='
                    | '<>' | '==') expression
                  ;
  argumentList = expression (',' expression)* ;
}
```

Table 4 Object Expression Language Expressions Grammar

This expression language does not include all parts of the original OCL, for example the ‘@pre’ construct is not included, neither are the constructs for sending messages. These omissions could be added to this grammar, or added to an extension of it.

3.2 An Object Constraint Language (OCL)

Now that we have defined a basic object expression language, we can reuse this syntax to aid in the definition of other languages based upon it. We first show a specification for an object constraint language in Table 5.

```
package ocl;

grammar constraints extends oel.expressions {

  constraint = typeConstraint | operationConstraint ;
  typeConstraint = invariant | defintion ;
  invariant = 'context' type 'inv' ':' expression ;
  definition = propertyDefinition | operationDefinition ;
  propertyDefinition = 'context' type 'def ':'
                        variableDeclaration '=' expresison ;
  operationDefinition = 'context' type 'def ':'
                        operationSignature '=' expresison ;
  operationConstraint = 'context' operationSignature
                        ('pre'|'post'|'body') ':' expression ;
  operationSignature = pathName '(' variableDeclarationList ')'
                       ':' type ;
}
```

Table 5 Object Constraint Language Grammar

3.3 An Object Query Language (OQL)

An object query language expression is simply an object expression in the context of a number of free variables; Table 6 illustrates a grammar specification.

```
package ocl;

grammar queries extends oel.expressions {

  query = 'context' variableDeclarationList
          'query' ':' expression
}
```

Table 6 Object Query Language Grammar

3.4 An Object Action Language (OAL)

To provide an object action language, it is necessary to provide mechanisms to create and destroy objects and to assign values to properties. We can extend the object expression language to add these mechanisms as shown in grammar of Table 7.

```
package oal;

grammar actions extends oel.expressions {

  actionList = (action ';')+ ;
  action = constructObject | destroyObject
         | assignment | variableDefinition ;
  constructObject = 'create' type
                    '{' variableDefinitionList '}' ;
  destroyObject = 'destroy' expression ;
  assignment = navigationExpression ':=' expression ;
}
```

Table 7 Object Action Language Grammar

4 Conclusion

Many improvements to the OCL have been suggested in the literature and we believe they are required in order to improve the language. In addition, since OCL was originally developed, its use has begun to exceed its original purpose for specifying constraints, and is now used as a basis for a number of different languages. We suggest that this be more officially recognised in the OCL standard, perhaps by altering its name, or at least by discussing its use in a family of languages. In particular much of the specification of OCL can be altered to improve the easy by which it is reused as the core for other languages.

In addition to altering the specification of the OCL, improvements should be targeted at tools that provide support for new languages, in order for the tools to support the notion of families of languages. This paper has illustrated a possible mechanism for defining the concrete syntax of a family of languages, using an EBNF like syntax. Tool support for this grammar specification approach has been built and is currently being tested for efficiency and flexibility. We are also investigating mechanisms for providing extensible semantic mappings based on the notion of model transformations. A more complex extension to OCL is illustrated in [5].

References

[1] Akehurst D. H., "Relations in OCL," in proceedings UML <<2004>> Workshop: OCL and Model Driven Engineering, Lisbon, Portugal, October 2004.

[2] Akehurst D. H., "Validating BPEL Specifications Using OCL," University of Kent at Canterbury, technical report: 15-04, August 2004.

[3] Akehurst D. H. and Bordbar B., "On Querying UML data models with OCL," in proceedings <<UML>> 2001 - The Unified Modeling Language: Modelling Languages, Concepts and Tools, Springer, LNCS 2185, October 2001.

[4] Akehurst D. H., Howells W. G., and McDonald-Maier K. D., "Kent Model Transformation Language," in proceedings MoDELS 2005 Workshop Model Transformations in Practice, Montego Bay, Jamaica, October 2005.

[5] Akehurst D. H., Howells W. G., and McDonald-Maier K. D., "Kent Model Transformation Language," in proceedings Model Transformations in Practice Workshop, part of MoDELS 2005, Montego Bay,Jamaica, October 2005.

[6] Akehurst D. H., Linington P. F., and Patrascoiu O., "OCL 2.0: Implementing the Standard," University of Kent, Canterbury 12-03, November 2003.

[7] Akehurst D. H. and Patrascoiu O., "OCL 2.0 – Implementing the Standard for Multiple Metamodels," in proceedings UML 2003 Workshop, OCL 2.0 - Industry standard or scientific playground?, San Francisco, USA, October 2003.

[8] Cariou E., Marvie R., Seinturier L., and Duchien L., "OCL for the Specification of Model Transformation Contracts," in proceedings <<UML>> 2004 Workshop OCL and Model Driven Engineering, Lisbon, Portugal, October 2004.

[9] Demuth B., "The Dresden OCL Toolkit and its Role in Information Systems Development," in proceedings 13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice Conference, Advances in Theory, Practice and Education (ISD'2004), Vilnius, Lithuania, September 2004.

[10] Gentleware, "Poseidon UML tool," www.gentleware.org, 2003.

[11] Haustein S. and Pleumann J., "OCL as Expression Language in an Action Semantics Surface Language," in proceedings <<UML>> 2004 Workshop OCL and Model Driven Engineering, Lisbon, Portugal, October 2004.

[12] IBM, "Eclipse Universal Tool Platform," 2001, http:/www.eclipse.org

[13] Klasse-Objecten, "Octopus: OCL Tool for Precise Uml Specifications," 2005, http://www.klasse.nl/english/research/octopus-intro.html

[14] Parr T., "ANTLR Parser Generator," 2005, www.antlr.org

[15] Patrascoiu O. and Rodgers P., "Embedding OCL Expressions in YATL," in proceedings <<UML>> 2004 Workshop OCL and Model Driven Engineering, Lisbon, Portugal, October 2004.

Generation of an OCL 2.0 Parser

Birgit Demuth[1], Heinrich Hussmann[2], and Ansgar Konermann[3]

[1] Technische Universität Dresden, Department of Computer Science
`bd1@inf.tu-dresden.de`
[2] Ludwig-Maximilians-Universität München,
Faculty of Mathematics, Computer Science and Statistics
`heinrich.hussmann@ifi.lmu.de`
[3] Technische Universität Dresden, Department of Computer Science
`ansgar.konermann@gmx.de`

Abstract. The OCL 2.0 specification defines explicitly a concrete and an abstract syntax. The concrete syntax allows modelers to write down OCL expressions in a textual way. The abstract syntax represents the concepts of OCL using a MOF compliant metamodel. OCL 2.0 implementations should follow this specification. In doing so emphasis is placed on the fact that at the end of the processing a tool should produce the same well-formed instance of the abstract syntax as given in the specification. This offers the possibility to implement OCL-like languages with the same semantics that are for example easier to use for business modelers. Therefore we looked for a parser technique that helps us to generate an OCL parser to a large extent. In this paper we present the technique we developed and proved within the scope of the Dresden OCL Toolkit. The resulting Dresden OCL2 parser is especially characterized by using a generation approach not only based on a context-free grammar but on an attribute grammar to create the required instance of the abstract syntax of an OCL expression.

1 Introduction

The OCL 2.0 specification [1] defines explicitly a concrete and an abstract syntax. The abstract syntax represents the concepts of OCL using a MOF compliant metamodel. In the following, this model is also referred to as *OCL metamodel*. The definition of such a *model-based abstract syntax* is often used for modeling languages [2, 3]. The concrete syntax allows modelers to write down OCL expressions in a textual way. OCL 2.0 implementations should follow the OMG specification. In doing so emphasis is placed on the fact that at the end of the processing of an OCL expression a tool should produce the same well-formed instance of the abstract syntax as given in the specification. This offers the possibility to relatively cost-efficient implement OCL-like languages that are for example easier to understand and write for business modelers. In [4], an example of a *Business Modeling Syntax for OCL* is described. Implementing a parser by hand however is a tedious and error-prone process. Thus, instead of manual implementation, all parts of a parser should be *generated* from one or more formal specifications. Therefore we developed a parser technique that helps us to generate an OCL parser to a large extent. This includes the generation not only based on a context-free grammar but on an attribute grammar to create the required instance of the abstract syntax of an OCL expression. As a result of this technique an *extended SableCC* version of the well-known parser generator SableCC [5] for lexical and syntactical analyzers has been developed.

Besides the concrete syntax a further subject of variability are the UML and MOF metaclasses that are used in the OCL metamodel. The current OCL 2.0 specification refers to the UML 1.4 metamodel. In future the OCL metamodel has to be aligned with the UML 2.0 or other metamodels. Therefore we extended our generation approach to the overall compiler architecture as explained in [6, 7]. All metamodel classes are implemented by the JMI (JavaTM Metadata Interface [8]) based generation of their Java Interfaces. Metamodels can be incorporated by XMI files.

To the best of our knowledge our approach currently provides maximal flexibility and high productivity in the OCL 2.0 parser construction process. We developed and proved the parser generation approach within the scope of the Dresden OCL Toolkit [9]. Our solution differs from comparable implementations such as Octopus [10] by providing a clean separation of code which *computes* inherited and synthesized attributes, and code which performs tree walking and attribute *passing* to and from nodes. This separation eases implementation of semantic analysis, since the implementor is not required to deal with the tedious and error-prone task of attribute handling. Instead, a clean and elegant API to the attribute evaluator skeleton is provided. In addition, this approach simplifies the development of the generator for the attribute evaluator skeleton. The approach also facilitates maintenance of the attribute evaluator in case of changes or extensions to the OCL language, since large parts of the implementation can be generated from an L-attribute grammar. Due to this separation, our solution is superior to those ones mixing tree walking, attribute handling and attribute computation, as it is often the case with compiler generators employing semantic actions to specify semantic analysis.

In the following, we analyze the current OCL 2.0 specification and point out its major problems with respect to automatic parser construction (Section 2). In Section 3 we propose solutions for them. The main part of the paper (Section 4) explains how we implemented our parser generation approach in the Dresden OCL2 Toolkit (the reengineered Dresden OCL Toolkit). In Section 5 we summarize the results and the experience with the parser construction process. We also give an outlook on further development plans within the scope of the Dresden OCL2 Toolkit.

2 Deficiencies of the OCL 2.0 Concrete Syntax

The concrete syntax of OCL 2.0 exhibits some properties which complicate automatic parser construction. This section names those properties and explains why they make parser construction difficult.

Mixed recognition stages. The concrete syntax specification mixes specification means for all three stages of a parser, that is lexical, the syntactical and the context-sensitive analysis. For example, there is no precise definition of names.[1] Thus, a parser built based on this grammar would not take full advantage of the capabilities provided by the lexical analysis stage.

Instead, valid names are often only recognized by taking context information into account.[2] This also makes syntactical analysis context-sensitive, which is hard to handle with efficient parsing algorithms. Names can easily be recognized using regular languages, so the current approach of the specification easily leads to complex, inefficient parsers. As another example, valid binary operators are defined using disambiguation rules, typically executed during context-sensitive analysis. This prohibits successful analysis of binary expressions during the syntactical analysis stage.

In summary, the specification overstrains syntax analysis while underutilizing lexical analysis and using context-sensitive analysis inefficiently.

No analytic grammar. As above explained, the specification is structured around concepts of the *abstract syntax*. For the vast majority of elements of the abstract syntax, the specification contains one *production*, potentially consisting of more than one *alternative*. Each alternative defines a language making up valid textual representations of the abstract syntax element. These languages often bear little to no syntactic similarity. As an example, consider productions *IteratorExpCS* or *AttributeCallExpCS*. For parsing, it is important to group languages which are syntactically *similar*. This reduces the risk of parsing conflicts and is often the key to rendering efficient deterministic parsing possible.

Ineffective disambiguation. On the other hand, alternatives which *do* bear syntactic similarity are scattered across the specification. For each ambiguous production, a set of disambiguation rules is given, intended to disambiguate the regarding productions. The spatial dispersion of the rules makes it difficult to check whether they really make the grammar unambiguous. Our analysis showed in a similar manner as in [11] that in many cases, they do not. Since the disambiguation rule sets do not impose a defined order of evaluation, each set of rules must separate the language of the regarding production from *all* other productions involved in the ambiguity. An explicit evaluation order would alleviate the situation, since it introduces an additional implicit rule into each rule set except the first, ensuring that all previous rule sets did not match.

No model of input artifacts. The transformation from concrete to abstract syntax is specified in terms of an attribute grammar. Attribute evaluation and syntactic disambiguation usually take place during context-sensitive analysis and are specified on top of the concrete syntax. The concrete syntax tree hence is

[1] of classifiers, attributes etc.

[2] namely, checking whether a name exists in the model

an input artifact for the attribute evaluator. To allow for concise specification of context-sensitive analysis, an explicit model of the concrete syntax is required. The specification does not define an explicit model of the concrete syntax, leaving derivation of the eventually existing implicit model to the user of the specification. This situation is unsatisfactory, as it involves guessing. It is especially true in the presence of inconsistent use of this model. This manifests itself in the specification in different notations for referring to the abstract syntax tree node *ast* or the inherited attribute *env.* Most attribute evaluation rules use *ProductionName.ast* to denote the current AST node, but TupleLiteralExpCS uses *tuplePart*, some attribute evaluation rules also use *ProductionName.* Most rules use *ProductionName.env* to refer to the inherited attribute, but VariableExpCS uses *env.lookup().*

Not machine-readable. The specification of the concrete syntax is provided by the OMG as a PDF file. Although being *machine-readable* in a narrower sense, the file format can neither be understood by parser generators, nor be easily transformed into an appropriate format automatically. The relevant parts of the specification have to be extracted and converted into the desired input file format *manually.* Older versions of the OCL included a link to a machine-readable grammar of OCL [12] (Chap. 6.9). To alleviate automatic parser generation, future versions of the specification should again include links to machine-readable versions of the concrete syntax.

3 Overcoming the Limitations of the Concrete Syntax

During the development of the OCL 2.0 parser for the Dresden OCL2 Toolkit, we encoutered the problems described in Section 2. This section explains the measures we took to solve or circumvent those problems.

Separate specifications for each transformation stage. From the OCL 2.0 specification, distinct specifications of lexical and syntactical language structure were derived, in essence manually. Both were written in SableCC [5] syntax. This allowed for subsequent testing of completeness and absence of parsing conflicts by simply feeding the specifications to the generator. The disambiguation and attribute evaluation rules were first dropped completely and re-introduced later.

Removal of syntactical ambiguities. The resulting grammar exhibited numerous parsing conflicts, resulting from syntactic ambiguities. It can be proved that no algorithm exists which computes for any given context-free grammar whether it is ambiguous or not ([13], Chap. 9.10). To allow for systematical removal of ambiguities, it is useful to construct the appropriate LR(k) automaton and remove any parsing conflict. If no parsing conflicts exist, the grammar is guaranteed to be unambiguous. We followed this algorithm to iteratively remove conflicts. Whenever a run of the SableCC parser generator revealed parsing conflicts, we modified the grammar by merging ambiguous productions. The language described by a grammar thus modified is usually larger than the one described by the original grammar. To keep the recognized language identical, all merged productions were noted. During context-sensitive analysis, they need to be differentiated and sentences not allowed by the original grammar need to be sorted out. This quickly led to a grammar partially resembling the OCL 1.x grammar quite closely [12]. Recognizing this, parts of the OCL 1.x grammar and the grammar from our older Dresden OCL Toolkit [14] were used as references during the remaining process of grammar restructuring.

This step resulted in an analytical LALR(1) grammar representing OCL 2.0, available in SableCC syntax and ready for automatic parser generation. It is a firm basis for definition of context-sensitive analysis.

Introducing an explicit model of the concrete syntax. Thanks to the use of SableCC, an explicit model of the concrete syntax comes for free. SableCC is capable of generating an object-oriented framework of classes representing the syntax tree. The transformation from grammar to framework classes is explicitly defined ([15], Chap. 5). Using the API of the framework classes, we can access each node of the syntax tree in a well-defined manner, including navigation to child and parent nodes as well as retrieval of token texts.

Redefining context-sensitive analysis. Having modified the OCL 2.0 grammar heavily, the disambiguation and attribute evaluation rules from the specification did not fit the new grammar any more. A new definition of the context-sensitive analysis stage had to be derived by hand. We stipulated that the resulting attribute grammar should be an L-attribute grammar [16]. This allows attribute evaluation to be performed in a single depth-first, left-to-right tree walk. We divided the context sensitive analysis stage into two alternating substages. The first one performs the walk over the concrete syntax tree, automatically passing attributes up and down in the tree. By default, it automatically creates ASM node instances for each synthesized attribute. It calls hook methods pertaining to the second substage whenever computation of attribute values or disambiguation is required. The specification of the first substage was incorporated into the tailored SableCC grammar, allowing complete generation of implementation code for this substage. The second substage comprises of implementations for the hook methods. These were implemented manually.

4 The OCL Parser of the Dresden OCL2 Toolkit and Its Generation Process

The *Dresden OCL Toolkit* is a well-established software package [9] providing OCL support, either through standalone tools or through libraries which can be integrated into tools by third parties. It has been developed at the Technische Universität Dresden and underwent a reengineering process to accomodate it to the new OCL 2.0 standard, with the new version called *Dresden OCL2 Toolkit.*

This section describes the build process used to create the OCL2 parser of the Dresden OCL2 Toolkit. It sketches important aspects of the parser's architecture, followed by a detailed explanation of the features of both a SableCC extension used to generate an attribute evaluator skeleton and of the resulting parser implementation. A few examples illustrate the features.

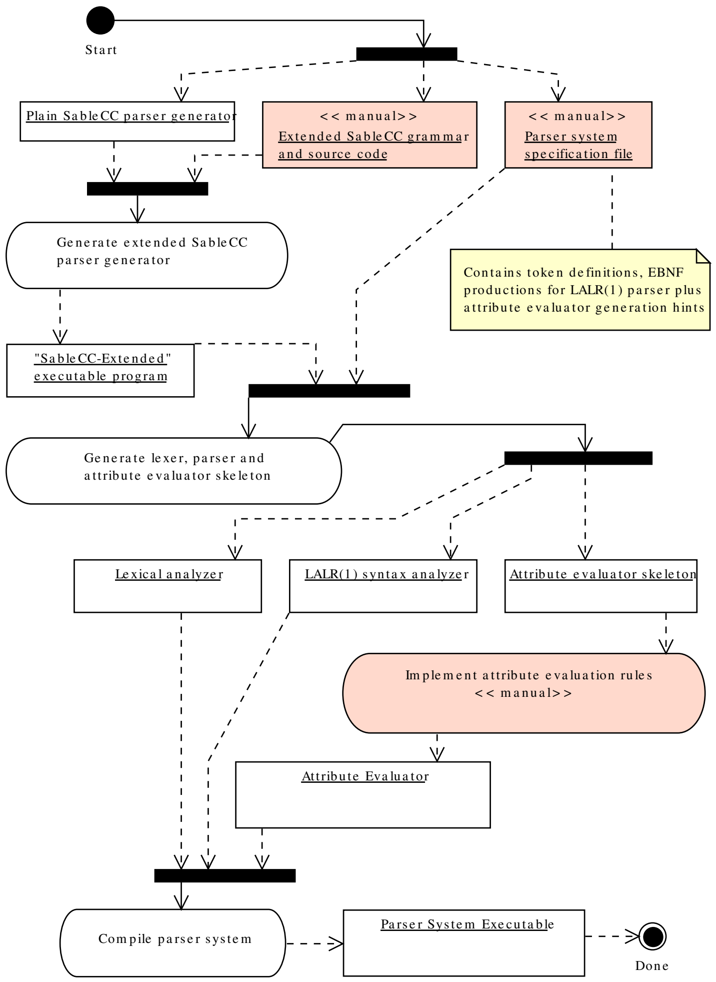

Start
Plain SableCC parser generator
<< manual>> Extended SableCC grammar and source code
<< manual>> Parser system specification file
Generate extended SableCC parser generator
Contains token definitions, EBNF productions for LALR(1) parser plus attribute evaluator generation hints
"SableCC-Extended" executable program
Generate lexer, parser and attribute evaluator skeleton
Lexical analyzer
LALR(1) syntax analyzer
Attribute evaluator skeleton
Implement attribute evaluation rules << manual>>
Attribute Evaluator
Compile parser system
Parser System Executable
Done

Fig. 1. Generation process used to generate the OCL 2.0 parser

4.1 Generation Process

The process employed to generate the parser is outlined by the activity diagram given in Figure 1. Activities and objects tagged *manual* are performed or created by hand, the remaining ones automatically. The process requires the following tools and input artifacts:

- an unmodified instance of the open-source compiler generator SableCC, version 2.18.1 [5]
- an extended version of SableCC, consisting of
  - a SableCC grammar for *extended* grammar files
  - enhanced Java source code making SableCC capable of parsing and using extended grammar files
- an OCL 2.0 grammar in extended syntax
- an attribute evaluator implementation for OCL 2.0

Building the parser system involves three major steps. First, an extended version of SableCC has to be created, allowing it to act as a *generator* for the context sensitive analysis stage implementation. During the next step, the extended SableCC is used to generate a lexical and syntactical analyzer for OCL 2.0, plus an attribute evaluator skeleton. This is an abstract base class and has to be implemented according to the OCL 2.0 specification to derive a working attribute evaluator.

4.2 The Parser Architecture

The resulting parser uses two passes to transform the input text into an Abstract Syntax Model (ASM). The ASM represents an instance of the OCL metamodel as defined by the OCL 2.0 specification. During the first pass, it *constructs* the concrete syntax tree (CST). This is performed automatically by the SableCC-generated parser code. In the second pass, an attribute evaluator transforms the CST into an ASM. The attribute evaluator makes use of a modified visitor design pattern [17] called *tree walkers*, as do all of the tree walker classes generated by SableCC ([15], Chap. 6).

The attribute evaluator is designed to successfully perform attribute evaluation for any L-attribute grammar. Thus it performs a single-sweep, depth-first, left-to-right tree walk. All visit methods now take one additional argument, representing the inherited attributes. The synthesized attributes are passed to their parent node as return value.

The parser is integrated into the existing metamodel-based OCL 2.0 compiler architecture of the Dresden OCL2 toolkit [6, 7]. All ASM nodes are created using a variant of an abstract factory [17], which in turn uses the API exposed by the compiler architecture to create instances of appropriate OCL 2.0 metamodel elements.

4.3 Features of the Extended SableCC Generator

Implementing tree walking code for the context-sensitive analysis stage by hand was soon identified as a tedious and error-prone task. This is even more true since tree *walking* alone is not sufficient for an attribute evaluator. It must pass inherited attribute values into child nodes and collect synthesized attributes produced for child nodes. Besides, it must perform *computation* of attribute values, observing data dependencies between them. Finally, all synthesized attributes must be *stored* for later use, and possibly passed to attribute evaluation code for sibling child nodes.

Lacking a generally available generator for efficient attribute evaluators in Java, we decided to add some simple attribute evaluation features to SableCC 2.18.1. We also considered using SableCC 3.x and its built-in CST-to-AST transformation techniques. It supports transformation of a CST into an AST made up of classes *generated* from appropriate grammar productions. Since we needed to use our own JMI-based ASM classes, we quickly discarded this approach. Besides, SableCC 3.x does not support using context information during transformation from CST to AST. This feature is however elementary for OCL 2.0, since many transformations depend on model information and are thus context-sensitive.

The section motivates the need for a more elaborate tree walking code than currently generated by SableCC. It illustrates the main features of our SableCC extension.

Support for attribute handling. SableCC allows generation of simple tree-walker classes based on the visitor design pattern [15]. They allow calling of custom code at the beginning and end of a visit method for each node, but nowhere in between. This is not sufficient for L-attribute grammars, where inherited attribute values may need to be computed between sibling child nodes. The code does not store the ASM nodes computed during attribute evaluation of child nodes. The only possibility to achieve this using original SableCC are two generic maps, called *in* and *out*. They are intended to hold data items associated with ASM nodes, the ASM node objects acting as map keys. If using this facility, code to store and retrieve objects from these maps, as well as any type casting from java.lang.Object, needs to be written by hand. This is error-prone and should be prevented.

Figure 2 shows the grammar extract we use to describe OCL let expressions. It is written in extended SableCC syntax. Words in angle brackets, exclamation marks and keywords starting with a hash sign (#) are part of the syntax extensions, which can be ignored for now.

Figure 3 shows tree walking code generated by the original SableCC for the production introduced in Figure 2. Custom code can be utilized in line 2 and 13 by overriding methods $\{in|out\}$*ALetExpCs*. For let expressions however, the OCL 2.0 specification stipulates that all variable declarations must be passed to the attribute evaluation code of the body expression as part of the inherited attribute *env*. This means that somewhere between line 7, where attribute evaluation for the variable declarations occurs, and line 11, where the same occurs for the body expression, custom code to compute the correct value for *env* must be incorporated. Besides, the generated code does allow neither passing inherited attribute values *into* the attribute evaluation method, nor passing synthesized attribute values *back* to the parent node. Please also note that the code descends

```
1  Tokens
2    ! in  = 'in';
3    ! let = 'let';
4  Productions
5    let_exp_cs <LetExp> =
6        let [variables]:initialized_variable_list_cs
7        in [expression]:expression #customheritage
8      ;
```

Fig. 2. Grammar extract describing *let expressions*

into child nodes representing irrelevant syntactic sugar, like the 'let' token. This is a waste of computational resources and should be prevented.

```
public void caseALetExpCs(ALetExpCs node) {
  inALetExpCs(node);
  if(node.getLet() != null) {
    node.getLet().apply(this);
  }
  if(node.getVariables() != null) {
    node.getVariables().apply(this);
  }
  // ...
  if(node.getExpression() != null) {
    node.getExpression().apply(this);
  }
  outALetExpCs(node);
}
```

Fig. 3. Original SableCC tree walker code traversing a *let expression* node (simplified)

Figure 4 shows the code generated by our extended SableCC. First note that an additional parameter *param* was introduced, which is automatically casted to type *Heritage*. This type is a data container comprising all inherited attributes ever required during the tree walk, including *env*. Attributes not used in a specific context contain null values. The return type of the visit method is now *LetExp*, as specified on line 5 of Figure 2. The ASM nodes of child nodes are returned by their visit methods (lines 8, 17) and *stored* in a variable. This even works for *lists* of CST nodes, which are converted to lists of their ASM nodes.

Computing inherited attributes. Incorporation of custom code for computation of inherited attributes is performed on demand if the corresponding production element is followed by the keyword *#customheritage*, as it is the case for *expression* in Figure 2. The generated code in Figure 4 (lines 14-15) calls an ab-

```
public final LetExp caseALetExpCs(ALetExpCs node, Object param) {
  Heritage nodeHrtg = (Heritage) param;
  Heritage childHrtg = null;

  PInitializedVariableListCs childVariables = node.getVariables();
  List astVariables = null;
  if(childVariables != null) {
    astVariables = (List) childVariables.apply(this, nodeHrtg.copy());
  }

  PExpression childExpression = node.getExpression();
  OclExpression astExpression = null;
  if(childExpression != null) {
    childHrtg = insideALetExpCs_computeHeritageFor_Expression(node,
      childExpression, nodeHrtg.copy(), astVariables);
    // ...
    astExpression = (OclExpression) childExpression.apply(this,
      childHrtg);
  }

  LetExp myAst = (LetExp) factory.createNode("LetExp");
  myAst = computeAstFor_ALetExpCs(myAst, nodeHrtg,
    astVariables,
    astExpression);
  return myAst;
}
```

Fig. 4. Tree-walker code generated by extended SableCC traversing a *let expression* node (simplified)

stract method *inside<Alternative>_computeHeritageFor_<Node>*. The parameter list of this method has a variable length, determined at generator run-time. It contains the current CST node, the CST node of the child we are about to visit, a copy of the current heritage, and the ASM nodes of all left siblings. The latter is required to fully support L-attribute grammars.

Skipping irrelevant child nodes. The code in Figure 4 does *not* descend into all child nodes given in the grammar production. The exclamation mark in front of a token definition or a production element prevents the attribute evaluator generator to create code for *irrelevant* nodes, such as tokens merely used as syntactic markers (e. g. 'if', 'let', etc). This can save some computational resources.

Creation and computation of ASM nodes. *Creation* of ASM nodes is performed automatically by default, using a factory (Figure 4, line 21). This can be switched off on demand. *Computation* of the ASM node's member values is delegated to an abstract method called *computeAstFor_<AlternativeName>*. This method is basically responsible for computation of synthesized attributes according to the attribute evaluation rules defined in [1]. Again, the parameter list is variable, allowing not only to pass the ASM node to be initialized and the current Heritage, but also the ASM nodes of *all* left sibling nodes. This is required to support L-attribute grammars. Besides, the implementation can take context information into account, obtained as inherited attribute *Heritage*, to perform proper disambiguation.

Automatic node creation can be switched off for each element of a production by appending the keyword *#nocreate*. This will result in a slightly modified signature for the corresponding *createAstFor_Xxx* method. The feature is particularly useful for efficient conversion of recursively defined lists into their equivalent ASM counterparts.

As an example, the relevant grammar extract for context declaration lists is shown in Figure 5. The grammar recursively describes a simple list of context declarations. Figure 6 shows the corresponding generated code. The ASM node type for a context declaration list is a *List* of OclContextDeclaration instances, a type specific to our implementation. It is not defined in the OCL 2.0 abstract syntax.

```
context_declaration_list_cs <List> =
    [context]:context_declaration_cs
    [tail]:context_declaration_list_cs? #nocreate
  ;
```

Fig. 5. Grammar extract for context declaration lists

```
 1  public final List caseAContextDeclarationListCs(...) {
 2    if( childContext != null) {
 3      astContext = (OclContextDeclaration) childContext.apply(...);
 4    }
 5    if( childTail != null) {
 6      astTail = (List) childTail.apply(this, nodeHrtg.copy());
 7    }
 8    List myAst = computeAstFor_AContextDeclarationListCs(nodeHrtg,
 9      astContext, astTail);
10    return myAst;
11  }
```

Fig. 6. Generated code for context declaration lists (simplified)

In contrast to Figure 4, where the ASM node is created just before the call to the *computeAstFor* method (line 21), there is no such call in Figure 6 (line 7-8). Instead, the responsibility to create the ASM node is delegated to the *computeAstFor* method (line 8-9). This method will be called for the last context in the input text *first*, since the tree walker performs a depth-first descent and all child nodes are evaluated first, including potential list tails. Thus, the skeleton code allows for the attribute evaluation code of the last context declaration to create a list instance containing the ASM couterpart for the currently processed node. All preceding context declarations can then be added to the head of this list. Figure 7 shows the actual implementation code for this example.

```
public List computeAstFor_AContextDeclarationListCs(
    Heritage nodeHrtgCopy, OclContextDeclaration astContext,
    List astTail)
{
  List result = null;
  if ( astTail != null ) {
    astTail.add(0, astContext);
    result = astTail;
  } else {
    result = new LinkedList();
    result.add(astContext);
  }
  return result;
}
```

Fig. 7. Attribute evaluation code for context declaration lists (simplified)

Automatic attribute passing for chain rules. The concrete syntax specification contains numerous productions comprising alternatives of the form $A \rightarrow B$, with A and B being nonterminals. Such rules serve to subsume various syntactical instances of a generic semantic concept under a common production, delegating definition of the actual syntax to subordinate productions. We call these productions *chain rules.* One example in our modified grammar is *literal_exp_cs* (Fig. 8).

The attribute evaluator code can be simplified for this type of productions. It is not necessary to compute an ASM node, if the ASM node type of subordinate, chained alternatives is conforming to the ASM node type of the embracing production. Our attribute evaluator generator supports this simplification through the keyword *#chain* appended to respective alternatives. The ASM node type for the embracing production is *LiteralExp*, which is a supertype of all ASM node types of the chained alternatives. The generator checks type conformance at generator run-time and issues an error message if the types do not match.

```
literal_exp_cs <LiteralExp> =
    {lit_collection}   collection_literal_exp_cs   #chain
  | {lit_tuple}        tuple_literal_exp_cs        #chain
  | {lit_primitive}    primitive_literal_exp_cs    #chain
  ;
```

Fig. 8. Production for literal expressions (simplified)

```
 1  public final LiteralExp caseALitCollectionL...(...) throws ... {
 2    // ...
 3    CollectionLiteralExp astCollectionLiteralExpCs = null;
 4    if(childCollectionLiteralExpCs != null) {
 5      astCollectionLiteralExpCs = (CollectionLiteralExp)
 6        childCollectionLiteralExpCs.apply(...);
 7    }
 8    LiteralExp myAst = astCollectionLiteralExpCs;
 9    return myAst;
10  }
```

Fig. 9. Generated code for chained alternative *lit_collection* of literal_exp_cs (simplified)

Figure 9 shows the generated code for this example. After descending into the (only) child node (lines 4-7), the visit method simply returns the ASM node created for the child (lines 8-9). This completely removes the need to implement ASM node computation manually.

4.4 Features of the Attribute Evaluator Implementation

Some features of the parser belong to the overall implementation and are not limited to the attribute evaluator. This section sketches them.

Balanced syntactical and semantic analysis for leaner implementation. During implementation of the attribute evaluator, the structure of the grammar was further modified to minimize implementation effort. The generator creates one visit method per alternative. Context-sensitive analysis for each alternative is to be performed in the corresponding visit method. In situations where it *is* possible to distinguish similar languages *syntactically*, care must be taken not to overuse this possibility. It can easily lead to a large number of alternatives describing nearly identical languages. By experience we learned that context-sensitive analysis for similar alternatives tends to require similar context-sensitive checks. This would result in sections of the same code in a large number of visit methods. To prevent code-duplication, we tried to balance exploitation of syntactical analysis and redundancy of context-sensitive analysis. Thus, some recognition effort was shifted from syntactical to contextual analysis, reducing the number of visit methods and thereby leading to a slightly leaner implementation.

Support for multiple iterator variables. According to [11], it is hard to implement multiple iterator variables syntactically using the grammar given in [1]. We were able to solve this problem using different syntactic constructs for variable declarations *with* or *without* initializer values, as assumed by [11].

5 Summary

We have learned that to generate an OCL 2.0 parser according to the OMG specification is a challenging task. Experimenting with the OCL 2.0 concrete syntax and changing it, we found a technique that allows to a large extent automated generation of a parser creating the ASM for a given OCL expression. The algorithm is based on a L-attribute grammar and has been implemented as extension of the SableCC parser generator. We plan to prepare this extension as *User Contributed Tool* to the Open Source Community ([5]). We implemented and tested the attribute evaluation based on an L-attribute grammar as part of the Dresden OCL2 Parser. Furthermore, our OCL2 Parser is integrated into the Dresden OCL2 Toolkit architecture. A first use case of the parser can be demonstrated by the OCL22SQL tool that generates SQL code as explained in [18]. We are currently starting a few new projects around the Dresden OCL2 Toolkit. Among other things we will investigate techniques for code generation of procedural/object-oriented (e.g. Java or C#) and declarative (e.g. SQL or XML query languages) code.

Acknowledgment. We would like to thank all people who have contributed over several years to the Dresden OCL Toolkit project. The project has been initiated in 1999. In the following years many students accounted both with research ideas and implementations to the Dresden OCL Toolkit and made their modules available to the open source community. Concerning the Dresden OCL2 Toolkit, Stefan Ocke created a solid basis for the new toolkit version by his Dresden OCL2 Repository that manages models and metamodels and has been an important prerequisite for the running Dresden OCL2 Parser.

References

1. Object Management Group: UML 2.0 OCL Specification. (2004) OMG Document ptc/2004-10-14.
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories. Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
3. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodelling. A Foundation for Language Driven Development Version 0.1. albini.xactium.com (2005)
4. Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition. Getting Your Models Ready for MDA. Addison-Wesley (2003)
5. Gagnon, E.M.: Sablecc parser generator. (www.sablecc.org)
6. Ocke, S.: Entwurf und Implementation eines metamodellbasierten OCL-Compilers. Master's thesis, Technische Universität Dresden, Department of Computer Science (2003)
7. Loecher, S., Ocke, S.: A Metamodel-Based OCL-Compiler for UML and MOF. Electr. Notes Theor. Comput. Sci. **102** (2004) 43–61
8. JCP: The JavaTM Metadata Interface (JMI) Specification. (www.jcp.org/en/jsr/detail?id=40)
9. Technische Universität Dresden, D.o.C.S.: Dresden OCL Toolkit. (dresden-ocl.sourceforge.net)
10. Klasse: Octopus: OCL Tool for Precise Uml Specifications. (www.klasse.nl/english/research/octopus-intro.html)
11. Akehurst, D.H., Patrascoiu, O.: OCL 2.0 - Implementing the Standard for Multiple Metamodels. Electr. Notes Theor. Comput. Sci. **102** (2004) 21–41
12. Object Management Group: Unified Modeling Language Specification Version 1.4.2. (2004) OMG Document formal/04-07-02, www.omg.org.
13. Grune, D., Jacobs, C.J.H.: Parsing techniques: a practical guide. Ellis Horwood, Upper Saddle River, NJ, USA (1990)
14. Finger, F.: Design and Implementation of a Modular OCL Compiler. Master's thesis, Technische Universität Dresden, Department of Computer Science (2000)
15. Gagnon, E.: SableCC, an Object-Oriented Compiler Framework. Master's thesis, McGill University (1998)
16. Grune, D., Bal, H.E., Jacobs, C.J., Langendoen, K.G.: Modern Compiler Design. Wiley (2000)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
18. Demuth, B., Hussmann, H., Löcher, S.: OCL as a Specification Language for Business Rules in Data Base Applications. In Gogolla, M., Kobryn, C., eds.: UML 2001 - The Unified Modeling Language. 4th International Conference. LNCS 2185, Springer (2001)

Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java

Wojciech J. Dzidek[2], Lionel C. Briand[1,2], Yvan Labiche[1]

[1] Software Quality Engineering Laboratory, Department of Systems and Computer Engineering – Carleton University, 1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada
{briand, labiche}@sce.carleton.ca
[2] Simula Research Laboratory
Lysaker, Norway
dzidek@simula.no

Abstract. Analysis and design by contract allows the definition of a formal agreement between a class and its clients, expressing each party's rights and obligations. Contracts written in the Object Constraint Language (OCL) are known to be a useful technique to specify the precondition and postcondition of operations and class invariants in a UML context, making the definition of object-oriented analysis or design elements more precise while also helping in testing and debugging. In this article, we report on the experiences with the development of ocl2j, a tool that automatically instruments OCL constraints in Java programs using aspect-oriented programming (AOP). The approach strives for automatic and efficient generation of contract code, and a non-intrusive instrumentation technique. A summary of our approach is given along with the results of an initial case study, the discussion of encountered problems, and the necessary future work to resolve the encountered issues.

1 Introduction

The usefulness of analysis and design by contract (ADBC) has been recognized by current and emerging software paradigms. For example, in [1], a book on component software, an entire chapter is devoted to the subject of contracts, and the author argues that using a formal language to specify them would be ideal except for the disadvantage of the complexity associated with the usage of a formal language. However, recent experiments have shown that OCL provides a number of advantages in the context of UML modeling [2], thus suggesting its complexity to be manageable by software engineers. Likewise in [3], a book discussing distributed object-oriented technologies, Emmerich argues that the notion of contracts is paramount in distributed systems as client and server are often developed autonomously. Last, model driven architecture (MDA), also known as model driven development (MDD), is perceived by many as a promising approach to software development [4]. In [4], the authors note that the combination of UML with OCL is at the moment probably the best way to develop high-quality and high-level models, as this results in precise, unambiguous, and consistent models. Having discussed the advantages of OCL, it comes as a surprise that the language is not used more widely for ADBC. One reason for this might be the well-established prejudices against any formal elements among software development experts and many influential methodologists. Another reason for the unsatisfactory utilization of OCL is the lack of industrial strength tools, e.g., tools to generate code assertions from OCL contracts.

The benefits of using contract assertions in source code is shown in [5], where a rigorous empirical study showed that such assertions detected a large percentage of failures and thus can be considered acceptable substitutes to hard-coded oracles in test drivers. This study also showed that contract assertions can be used to significantly lower the effort of locating faults after the detection of a failure, and that the contracts need not be perfect to be highly effective. Based on such results, the next step was therefore to address the automation of using OCL contracts to instrument Java systems. This paper reports on our experience with the development and use of *ocl2j*, a tool for the automated verification of OCL contracts in Java systems [6]. These verifications are dynamic, i.e., performed at run time, as opposed to offline – after the application executed.

The paper briefly starts with background information and related work. Then we go through an overview of our approach, followed by a discussion of some of the main technical and methodological issues with respect to transformation of constraints from OCL to Java. Next, an initial case study, aimed at showing the feasibility of the ocl2j approach, is presented. Finally, difficulties with using OCL for this purpose are outlined, conclusions are then provided.

2 Related Work

Currently, two tools exist for the purpose of dynamic enforcement of OCL constraints in Java systems: the Dresden OCL toolkit (DOT) [7, 8] and the Object Constraint Language Environment (OCLE) [9]. We decided to implement our own solution as DOT did not fulfill all of our requirements (as discussed below) and OCLE did not exist at the time and, after close examination, turned out not to fully address our needs.

Our aim was to have a tool that would: (1) support all the core OCL 1.4 functionality, (2) correctly enforce constraints, (3) instrument (insert the contract checking and enforcement code) program code at the bytecode level (as opposed to altering the source-code), (4) allow for optional dynamic enforcement to the Liskov Substitution Principle (LSP) [10], (5) support for separate compilation (i.e., allowing modifications of the application source code without recompiling assertion code or vice-versa), (6) correctly check constraints when exceptions are thrown, (7) have the ability for assertion code to use private members, (8) have the option to use either compile-time or load-time instrumentation (with load-time instrumentation constraint checking code can be installed or removed without requiring recompilation), and (9) have the ability to add assertions to classes for which the source-code is not available.

DOT was the pioneering work for this problem and is open-source software. It relies on the following technical choices. First, the instrumentation occurs at the source code level, requiring that the original program's source code be heavily modified. Original methods are renamed and wrapped, and supplementary code is added. The second choice is to check invariants only after methods that modify attributes used in invariants, instead of checking them before and after every public operation. The latter could indeed turn out to be inefficient when only a subset of methods in the class affect the invariant. In order to achieve this, DOT introduces a backup attribute for every attribute in every class, virtually cloning each object. It is then possible to check after a method execution whether it modified an attribute involved in the class invariant, i.e., whether the invariant should be checked. Another technical choice concerns the differences between OCL and Java types. DOT implements the OCL types in Java and wraps Java variables (attributes, method parameters or return value) used in assertions with equivalent OCL types. This results in additional objects created at runtime and more method calls (the resolution of the OCL to Java types occurs at runtime, this topic will be discussed in Section 4.2). Last, the generated code is constructed in such a way that it uses Java reflection mechanisms at runtime (i.e., when the instrumented program executes) to determine implementation details. The above technical decisions result in a large memory and performance penalty as a direct consequence of the virtual cloning (of all objects) and the wrapping (of all objects involved in OCL constraints). Support for OCL is also incomplete as, for example, query operations are not supported. Furthermore, constraints on elements in collections are not properly enforced as changes to elements in the collection can go unnoticed [6]. Additionally, since DOT inserts the contract code directly into the source code the user is faced with a dilemma: keep only the clean version or the instrumented one, or keep both versions of the source code. These options have disadvantages as when only one version of the code is kept then the user must deal with the long wait times for the cleaning and instrumentation whenever a change to the source needs to be made. If the version of the code being kept is the instrumented version then the code must be cleaned whenever the user wants to read the code. Keeping both versions of the source code solves some of those problems but introduces new ones. If a version management system is used, two versions of the source code must be kept in the system thus leading to inevitable inconsistencies. Last, there is no support for constraint inheritance.

OCLE is a UML CASE tool offering OCL support both at the UML metamodel and model level, though we only look at a portion of the tool: it's support for dynamic OCL constraint enforcement. Like DOT, instruments the source code and is limited in its support of OCL (e.g. the `@pre` keyword and the `oclIsNew()` operation are not supported). Furthermore, it cannot instrument existing source code (it only generates code skeletons from class specifications).

Note that although other tools exist that add design by contract support to Java [11, 12], they are not discussed in this paper as they do not address the transformation of OCL expressions into assertions.

3 The ocl2j Approach

This section presents our approach (ocl2j) towards the automatic generation and instrumentation of OCL constraints in Java. Our approach consists of Java code being created from OCL expressions and the target system then being instrumented: (1) The necessary information is retrieved from the target system's UML model and source code; (2) Every OCL expression is parsed, an abstract syntax tree (AST) is generated [7], and the AST is used to create the assertion code (the OCL to Java transformation rules were defined as semantic actions associated with production rules of the OCL grammar [13]. The generation of Java assertions from OCL constraints is thus rigorously defined and easily automated.); (3) The target system is then instrumented with the assertion code, using AspectJ which is the main Java implementation of Aspect Oriented Programming (AOP) [14, 15]. Note that since the techniques involved in step (3) are already covered in [16], this paper will only contain a brief overview of the strategy.

The ocl2j instrumentation is non-intrusive, as AOP technology is used to separate the assertions from the target system's source code. The user can work on the code without having to regenerate the contract assertions before each compile. This is true as long as the class diagram (including operation signatures and class relationships) and contracts do not change as then the contract code will not need to change either. Furthermore, this solution allows for parallel development of the target system's source code and its aspects, once again as long as the class diagram and contracts are stable. If the user wants to run the program without the contracts the user can recompile the code without including/weaving the aspects or chose not to weave the aspects into the bytecode at load-time if that instrumentation method is used. The weaving of the contract code and the target system's code is done at the byte code level. For this reason the source code is not necessary for the insertion of the contract code. This is very important if the target system uses classes for which the source code is not available.

Furthermore, instead of converting all variables in the Java target system's source code that participate in contracts to their OCL equivalent variables (through wrapping and cloning), a set of rules transform OCL types and operations to their equivalent Java types and operations. Note that in the ocl2j approach these transformations are performed when the target system is instrumented (as opposed to them being performed when the instrumented target system executes).

The section starts (Section 3.1) with a discussion of how OCL types are transformed in Java types. Next, Section 3.2 discusses the topic of equality with respect to OCL and Java. Section 3.3 shows how the OCL `@pre` construct is addressed. Section 3.4 provides rules on when contracts should be checked, with the corresponding discussion on how we do this with AspectJ in Section 3.5. Finally, after a brief introduction to AspectJ, Section 3.6 shows how we were able to use AspectJ to provide clean and efficient support for `oclAny::oclIsNew()`.

3.1 OCL to Java Transformations

The checking of contracts at runtime slows down the execution of the program. If this slowdown is too great the developers will not use the technology. For this reason it is important to focus on techniques that enable faster checking of contracts. One of these techniques is to translate OCL expressions directly into Java using the types retrieved from the target system (through reflection) at the assertion-code generation stage, instead of wrapping Java types and operations with OCL-like types and operations. The translation time is thus spent during instrumentation rather than execution. This distinction becomes critical during maintenance of large systems since changes to the system only occur to the subsystem under development. For this reason it is both unnecessary and inefficient to perform the OCL to Java type resolution over the whole system every time the system is executed.

Our OCL to Java type resolution relies on the following principles. First, whenever a simple mapping exists between OCL and Java types/operations, the translation is straightforward. For instance, OCL collection operation `size()` maps directly to the `size()` operation of the `java.util.Collection` interface (which every collection class in Java implements). When OCL types/operations cannot be directly converted to types/operations from standard Java libraries, the instrumentation code (aspect code) provides the functionality that is "missing" in the libraries. This ensures that no wrapping is necessary, and no additions to the target system are required. The aspect contains inner classes with operations that provide additional functionality to complete the mapping to Java such as the `collection->count(obj):Integer` operation, that counts the number of times object `obj` occurs in `collection` and does not have any counterpart in Java collection classes/interfaces. The aspect code thus contains inner class `OclCollection` with a `count` operation that takes two arguments: the collection on which `count` must be performed and the object that needs to be counted.

Next, OCL, unlike Java, has no notion of primitive types (e.g., `int`) as everything is considered an object. Java, on the other hand, supports primitive types and corresponding primitive value wrapper classes, or simply wrapper classes (e.g., `Integer`). OCL provides four, so-called, basic types: `Boolean`, `Integer`, `Real` and `String`. There is one exception to these differences in OCL and Java type systems: strings are objects in both OCL and Java. Having both primitive types and wrapper classes has a major impact on the process of transformation of OCL constraints into Java code[1]. For example, consider the following OCL constraint: `someCollection-> includes(5)`. When transforming the OCL expression into Java source code, 5 has to be transformed into either primitive value 5 or an instance of wrapper class Integer (`new Integer(5)`). As Java collections only take objects as elements, the latter is the correct choice.

A general, trivial solution to this problem would be to convert every literal value into an object, but as already discussed, this is inefficient. A more efficient solution consists in analyzing the types used in the OCL expression, the types required in the corresponding Java source code, as well as the characteristics of the expression, and converting objects to their primitive types when possible. During the transformation of OCL constraints into Java code the following strategy is followed:

- Values used in logical, addition, multiplication, and unary operations are evaluated in their primitive form.
- Values used as arguments in operation calls are converted, if necessary, into an instance of the required Java type according to the operation signature. E.g., the

[1] Note that this is only a problem for systems written in Java 1.4 and earlier as Java 1.5 has the autoboxing feature that addresses this problem.

java.util.Collection::contains(o:Object): boolean operation expects the parameter to be an object. Thus, in the case of the OCL constraint integerCollection->contains(10), where integerCollection is implemented in the Java source code as a Java collection (the class implements java.util.Collection), “10” is converted to an object using the integer object wrapper: new Integer(10). This would not be necessary if the expected parameter were of int type.

Taking a closer look at collection types reveals that OCL has three collection types, namely Set, Bag, and Sequence, whereas, Java only has two main collection interfaces, namely java.util.Set and java.util.List (we assume that the user-define collection implement java.util.Collection directly or indirectly). There is a direct mapping between OCL Set and java.util.Set and between OCL Sequence and java.util.List. However, OCL Bag does not have a direct Java counterpart. A bag is a collection in which duplicates are allowed [17]. java.util.Set cannot be used to implement an OCL Bag as it does not allow duplicates. The only possible alternative, which is assumed in the ocl2j approach, is to implement OCL Bag with java.util.List.

The three following situations are encountered when translating collection operations:

1. There is a direct mapping between an OCL collection operation and a java.util.Set or java.util.List operation, e.g., OCL operation includes() and Java operation contains().
2. The OCL collection operation does not have a direct counterpart but its functionality can easily be derived from existing java.util.Set or java.util.List operations. For instance, an implementation of OCL operation symmetricDifference() on Set can be built from operations removeAll() and addAll(). These transformations are performed by a specialized class within the aspect code, called OclCollection.
3. OCL collection operations that iterate over collections and evaluate an OCL expression (passed as a parameter to the operation) on elements in the collection are more complex. They do not have a direct Java counterpart and cannot be simply implemented using the operations provided by java.util.Set or java.util.List. These OCL operations are exists, forAll, isUnique, sortedBy, select, reject, collect, and iterate. They require more attention as the parameter is an OCL expression which requires to be instrumented as well in the aspect code. Templates and transformation rules are used to generate a unique method (residing in the aspect) for every distinct use of these operations.

Furthermore, note that transformations involving collections may also require that intermediate collections be generated in the assertion code for collections of objects that are used in OCL constraints but are not necessarily explicitly implemented as attributes in the code. This is the case with OCL collection operation coll->including(obj) that returns a collection with all the elements found in collection coll plus object obj. For instance, if an OCL expression shows coll->including(obj)->select(…), where select(expr) yields a subset of the collection (on which it is applied) containing all elements for which `expr` is true/satisfied. In that case, an intermediate collection containing all the elements of `coll` and `obj` has to be created before performing operation `select`.

3.2 Testing for Equality

Assertion code that tests for equality can take any one of three forms. First, if the values to be compared are of primitive type then the Java "`==`" construct is used in the equality test. Next, if the values being compared (or just one of them) are of reference type wrapping a primitive then the primitive value is extracted from the object using the appropriate method (e.g., `intValue()` for an object of type `Integer`) and again the values are tested for equality using the Java "`==`" construct. In other cases, objects are tested for equality using their `equals(o:Object):boolean` method. This is done as equality in OCL is defined at the object level, not the reference level. For example, let's take a look at the `java.awt.Point` class which has two attributes: `x:int` and `y:int`. Next, given two points `Point a = new Point(5, 5)` and `Point b = new Point(5, 5)`. If we compare these points at a "reference level" they will not be equal (`a == b` will evaluate to `false`), even though they the two objects `a` and `b` do represent the same point. Thus, `Point`'s `equals` method must be used to evaluate their equality (`a.equals(b)` does evaluate to `true`).

We assume that the `equals()` method is properly implemented [18] so that objects are deemed equal when their key attributes are equal. We define *key attributes* as attributes that define an object's identity (e.g., attributes `x` and `y` in the case of the `Point` class). Sometimes each instance of a class is unique (no clones are possible) in which case the default `equals()` functionality (i.e., inherited from `java.lang.Object`, considers each instance only equal to itself) will suffice as this functionality only compares reference values for equality, but when this is not the case the `equals()` method must be overridden. Note that this last point is often neglected by developers of Java-based systems.

3.3 Using Previous Property Values in OCL Postconditions

This section discusses the practical implementation of the OCL language construct `@pre`, used in postconditions to access the value of an object property at the start of the execution of the operation. Depending on the property that the `@pre` is associated with different values and amount of data must be stored temporarily until the constrained method finishes executing so that the postcondition can be checked. `@pre` can be used with respect to one of the following:

1. *Java types corresponding to OCL Basic types or query methods that return a value of such a type*. The mapping between these types is discussed in Section 3.1. In the case of a primitive type, the primitive value is stored in a temporary variable. In the case of an object, the reference to the object is stored in a temporary variable. Only

the reference is stored as these types are immutable and thus they cannot change (during the execution of the constrained method).

2. *Query methods that return an object*. In this case the objects are handled in the same way as described above, only the reference to that object is stored in a temporary variable (duplicated), the object itself is not cloned. The object is not cloned as we assume that the target system is written with proper encapsulation techniques, meaning that query methods that return an object to which the context class (the class containing the query method) is related via composite aggregation return a clone of the object, not the object itself. This is standard practice as discussed in Item 24 of [18]. Note that this is a necessary requirement as the following example will demonstrate: Consider a query method that returns the reference to an object `X` is queried, within the context of a postcondition, before a constrained method `M` executes (via the `@pre` keyword). During the execution of `M`, `X` is modified. Once `M` finishes execution the postcondition must be verified. Note that since the query method only returned a reference to `X` (instead of a clone of `X`), the postcondition will be evaluated with respect to the new version of `X`, as opposed the original version at precondition-time.
3. *Objects (references to objects)*. The object types in this discussion exclude the ones discussed in the points above. In this case a clone of the object is taken and stored in a temporary variable. We assume that the programmer properly implements cloneability support (as will be discussed).
4. *Collections*. A collection's identity is defined by the elements in that collection, thus a clone of a collection contains a clone of every element in the original collection. Using `@pre` on a collection will result in such a duplication of the collection in most cases. When the OCL collection operation being invoked on `someCollection@pre` is `size():Integer`, `isEmpty():Boolean`, `notEmpty():Boolean`, or `sum():T` then only the result of the operation is stored in the temporary variable. We note that in a lot of cases it may not be necessary to duplicate the collection in such a manner to enforce the postcondition correctly, but this is a subject for future work.

For a guide to providing support for cloneability see Item 10 in [18]. Essentially, two types of cloning methods exist. First, a shallow copy is where the fields declared in a class and its parents (if any) will have values identical to those of the object being cloned. In the case of a class exhibiting one or more composite relationships the shallow copy is not sufficient and a deep copy must be used—during a deep copy all the objects in the composition hierarchy must also themselves be cloned. To understand why, recall our objective here: We need access to the objects, as they were, before the constrained method executed. Objects are uniquely identified by their key attributes (key attributes are discussed in Section 3.2). If these objects have composite links to other objects (i.e., their class has composite relationships), thus forming a hierarchy of objects, the key attributes may be located anywhere in the hierarchy. A deep copy is therefore necessary.

3.4 Checking Contracts

Instrumenting a constraint requires that we identify where the corresponding assertion needs to be checked, the *insertion point*. The insertion point for an assertion checking a precondition is right before the execution of the corresponding method. Similarly, the insertion point for an assertion checking a postcondition is right after the execution of the corresponding method. As for class invariants, Meyer states that a class invariant must be true (i) after an instance creation and (ii) before and after any remote call to an operation [19]. In UML terms, this means that every public operation must be instrumented as public operations are the only ones that can participate to a remote call, according to Meyer's definition. However, a public operation in a UML model may end up being implemented as a public, package (default), or protected methods in Java, as package/protected methods are accessible to other classes in the same package[2]. In our approach, we decided to adhere to the UML definition: Only public operations as defined in the UML model are instrumented, whether those operations are implemented as public methods or not in the source code. This strategy is summarized in Table 1, which is adapted from [12].

Table 1. Constraint Checking

		public (UML)	not public (UML)	constructor
Pre	entry	X	X	X
Post	regular exit	X	X	X
	exception			
Inv	entry	X		N/A
	regular exit	X		X
	exception	X		

Table 1 also shows what is checked when an exception is thrown during the execution of a constrained method. A postcondition is not checked on abnormal termination of a method or constructor as the contract is likely not satisfied. The invariant is checked on abnormal termination of a public method execution since the object should still be in a legal state if the exception handling is correct. This is not the case for a constructor as an abnormal termination then suggests the construction of the object failed: checking the invariant is then not relevant.

This strategy for the instrumentation of assertions checking constraints is valid for non-static methods and constructors. Checking pre and postconditions for static methods can follow the same principles. Since the notions of invariants and constraint inheritance (following section) for static methods is not clearly defined in the literature, we decided to omit the check of constraints on static methods altogether in this paper, although further details can be found in [6].

[2] This differs from the UML definition of a protected operation which states that any descendant of the classifier can use the feature**.**

3.5 Instrumentation

Since our (non-OCL specific) instrumentation approach (for enforcing preconditions, postconditions, and invariants) is described in detail in [16], we will only provide a brief overview of our method here. Our instrumentation strategy relies on aspect-oriented programming (AOP) [14]. AOP is a methodology that facilitates the modularization of concerns in software development. In particular, it extracts cross-cutting concerns from classes and turns them into first-class elements: aspects. By decoupling these concerns and placing them in *aspects*, the original classes are relieved of the burden of managing functionalities orthogonally related to their purpose. Later, the aspect code is injected into appropriate places by a process known as weaving (usually at the program compilation stage).

Note that by using AOP as our instrumentation technology we were able to: (a) manipulate the bytecode instead of the source code; (b) be independent of coding conventions or extensions to the Java Virtual Machine (JVM); (c) optionally support LSP enforcement; (d) support separate compilation (i.e., allowing modifications of the application source code without recompiling assertion code or vice-versa); (e) support contract checking in the presence of exceptions; (f) have the ability for assertion code to use private members; (g) have the option to use either compile-time or load-time instrumentation (with load-time instrumentation constraint checking code can be installed or removed without requiring recompilation); and (h) have the ability to add assertions to classes for which the source-code is not available.

Let us describe, as an example, the structure of the aspect code for preconditions. Fig. 1 shows the code template (where strings in bold represent the variable parts of the aspect code) required to instrument the byte code. A `before` advice executes before the specified pointcut executes (i.e., the constrained method). The `before` keyword exposes variable names (with types) that can be used in the advice code: `self` of type `aClass`, and any method parameter (name and type) that the advice should use. (These will be used in the pointcut.) In the pointcut: **`execution(…)`** specifies, using a method signature, that any execution of method `aMethod` on any instance of class `aClass` is intercepted; **`target(self)`** maps `self` (defined in `before(…)`) to the object executing the intercepted method (on which the constraint is being evaluated). In the advice code, variable `self` will then be a reference to the object executing the intercepted method execution; **`args(…)`** maps names appearing in parenthesis (and defined in `before(…[params])`) to the parameters of `aMethod` so that its arguments can be referred to in the advice code;
`within(aClass)` specifies that the version of the executing method must be declared in class `aClass`. This is to prevent the interception of `aMethod`'s execution on subclasses of `aClass` that override `aMethod` (and thus likely have a precondition different from the one of `aMethod` in class `aClass`).

```
before(aClass self [, method parameters]):
  execution(method_return_type aClass.aMethod([method parameter
     types])) && target(self) [&& args(parameter names)]
     && within(aClass) {… //Check the precondition.}
```

Fig. 1. AspectJ code template for enforcing preconditions

In the following example, class `Person` has attributes `salary`, and `maxSalary`, all of type `Integer` in the UML model (`java.lang.Integer` in code). Additionally, class `Person` has an operation called `implementRaise(raise:Integer)` that raises the person's salary. The precondition for `implementRaise` is the following:

```
context Person::implementRaise(raise:Integer)
pre: self.salary + raise <= self.maxSalary
```

Fig. 2 shows the (incomplete) AspectJ code for the example, i.e., the aspect class in charge of checking the above precondition, following the template presented above.

```
privileged aspect Ocl2jAspect {
    // Before the execution of the implementRaise method in Person
    before(Person self, int raise) :
      execution(public void Person.implementRaise(int))
      && target(self) && args(raise) && within(Person) {
   if (!((self.salary.intValue()+raise)<=self.maxSalary.intValue())){
      constraintFailed("self.salary + raise <= self.maxSalary"); } }
   void constraintFailed(String constraint) {
      … // Report constraint failure. }
}
```

Fig. 2. AspectJ code for example precondition

3.6 `oclAny::oclIsNew()` Support

Any OCL type in a UML model, including user-defined classes, is an instance of `OclType`: it allows access to meta-level information regarding the UML model. In addition, every type in OCL is a child class of `OclAny`, i.e., all model types inherit the properties of `OclAny`. Among those properties is operation `oclAny::oclIsNew()` that can only be used in a postcondition: It evaluates to `true` if the object on which it is called has been created during the execution of the constrained method.

Java does not provide any functionality to which this operation could be mapped to. A traditional solution to the problem would be to instrument every construct that could be used to create the object of interest (e.g. all the constructors), though this solution would not be adequate in the case that the source code for the object of interest could not be modified.

The ocl2j solution to the problem of implementing operation `oclAny::oclIsNew()` is the following. If this operation is used on a type in any OCL expression, a collection is added to the aspect. This collection will store references to all the instances of the type created during the execution of the constrained method (as `oclAny::oclIsNew()` can only be used in the context of a postcondition): This is easily achieved with AspectJ as it only requires that the aspect comprises an advice to add, at the end of the execution of any constructor of the type of interest or its subtypes, the reference of the newly created instance. This raises the question of the choice of the Java data structure to store those references and the impact of aspect code on object garbage collection in Java: Objects in the instrumented program should be garbage collected if they are not used in the application code, even though they may be referenced by the aspect code. A solution to this problem is to use class `java.util.WeakHashMap` to store these references in the aspect. This was specifically designed so as to store references that would not be accounted by the garbage collector. It is based on a hash map where the keys are weak references to the objects we are monitoring. The garbage collector can get rid of an object, even when this object is still referenced, provided that these references are only used in instances of class `WeakHashMap`. When this is the case, the object is garbage collected and any reference to it removed from instances of the `WeakHashMap`.

Determining whether an object was created during the execution of the constrained method involves checking the `WeakHashMap` collection for the presence of the object in question. Finally, after the constrained method finishes executing and the postcondition is checked, the collection of instances (created during the execution of that method) is discarded.

Please note that this solution is not easily mapped to a solution that enables the use of the `oclAny:: allInstances()` construct as there is not way to *force* the JVM to run the garbage collection operation (though `Runtime.getRuntime().gc()` can be used to *suggest* this to the JVM). Thus, such an implementation of `oclAny:: allInstances()` could, in certain instances, return a collection of objects including ones that are designated for garbage collection (no longer referenced).

4 Preliminary Case Study

The case study is based on the system presented in [17]: The “Royal and Loyal” system example. Though modest in size, this system was chosen due to the large number of diverse constraints being already defined for it, including some quite complex ones. It should then provide initial evidence that ocl2j works for a wide variety of constraints. The UML model in [17] was expanded in this work to the system shown in [6] in order to be implementable. Once expanded, it was implemented in Java and consisted of 381 LOCs, including 14 classes, 47 OCL constraints, 53 attributes, and 46 operations.

The original version of the R&L system and the version with the assertion code (instrumented) are compared for a common scenario where a customer is added and makes purchases. We use the following three criteria for comparison: (1) bytecode size of the classes, (2) time it takes to execute the programs, (3) memory footprint. We only report in this paper the detailed results for (2) and provide a summary for (1) and (3).

Table 2 is based on the execution of five scenarios where the two main parameters affecting performance are varied from 1 to 100 (as indicated between brackets): (1) number of customers, (2) number of purchases over $25. Those two parameters affect the size of collections that are involved in the instrumented OCL contracts and are accessed during the execution of the program.

Our first observation was that all constraints were correctly translated into Java without encountering unexpected cases. Programs that have relatively large collections with many complicated constraints associated with these collections can expect, as a ballpark figure, a degradation in execution time of 2 to 3 times. Otherwise, the degradation in performance is smaller as the execution speed is slowed down by roughly 60%. This is significant but does not prevent the use of instrumented contracts in most cases during testing, unless the system's behavior is extremely sensitive to execution deadlines. The sources of degradation in performance have been further investigated in [6] where solutions are proposed for optimization. With respect to criteria (1), the target system grew 2.5 times in size, and (3), the maximum overhead percentage observed for the above scenarios were 14% and 10.5%, respectively.

Table 2. Execution time comparisons

Scenario	**Execution Time (ms)**		**Instrumented version slower by**
	Original	Instrumented	
1 (1,1)	38	62	63%
2 (10, 10)	41	64	56%
3 (100, 100)	47	111	136%
4 (1, 100)	39	76	94%
5 (100, 1)	47	106	125%

5 Future Challenges

While developing ocl2j we ran into several non-trivial issues that require significant work to address. Among others:

- The implementation of support for the `@pre` keyword leaves a lot of room for performance optimizations. For example, to evaluate the postcondition "`self.someCollection@pre = self.someCollection`" properly in every scenario one must create a new collection (say `tempCollection`) that holds a clone of every element present in `self.someCollection`. If `someCollection` is large or if the elements in that collection are expensive to clone, then the evaluation of this postcondition becomes very expensive. Furthermore, this potentially expensive operation is not even necessary if all the designer intended to check was whether `self.someCollection@pre` and `self.someCollection` point to the same object (i.e. hold the same reference). In such a situation the designer should be allowed to distinguish weather a deep or shallow copy is meant by the `@pre`. One way of addressing this would be by adding the keyword `@preShallow` to OCL.
- The use of `@pre` may lead to un-computable expressions. As shown in [20], the expression `self.b.c@pre` with respect to the example in Section 7.5.15 in [21] is not computable: "Before invocation of the method, it is not yet known what the future value of the `b` property will be, and therefore it is not possible to store the value of `self.b.c@pre` for later use in the postcondition!".
- Our experience revealed that, by far, the largest performance penalties (execution time overhead) of checking the OCL constraints during the execution of the system came from OCL collection operations [6]. For this reason we have started working

on an approach to minimize these performance penalties. In general the strategy involves checking a constraint on a collection whenever the state of the collection changes in such a way that it could invalidate the constraint. For example, consider the constraint `someCollection->forAll(someExpression)`. If this constraint is an invariant then it will be checked before and after any public method executes, even if neither the state of `someCollection` nor its elements changes. An alternative to this would be to check that `someExpression` holds for a newly added element to `someCollection`, and that `someExpression` holds for elements in the collection that undergo changes that may invalidate `someExpression`. This alterative will be more efficient on a large, often-checked, collection that does not undergo large changes. Note that this kind of strategy is facilitated by the use of AOP as the instrumentation technology.

- The implementation of the `OclAny::allInstances():Set(T)` functionality in Java is challenging since Java uses automatic garbage collection, i.e., objects do not have to be explicitly destroyed. Thus, the only way to know whether an object is ready to be garbage collected (and therefore not be in the `allInstances` set) is to run the garbage collection operation (costly execution-wise) after every state change in the system involving the destruction of a reference.

6 Conclusions

We have presented a methodology, supported by a prototype tool (ocl2j), to automatically transform OCL constraints into Java assertions and instrument these into the target program. The user of ocl2j can then specify whether a runtime exception is thrown or an error message is printed to the standard error output upon the falsification of an assertion during execution. This has shown, in past studies [5], to be extremely valuable during testing to detect failures and help debugging.

Transformation rules to translate OCL constraints into Java assertions have been derived in a systematic manner with the goal that upon instrumentation the generated assertion code will be efficient in terms of execution time and memory overhead. This was largely achieved thanks to the systematic definition of efficient semantic actions on production rules in the OCL grammar, and the minimization of reflection use at runtime.

The instrumentation of those Java code assertions is performed by employing aspect-oriented programming (AOP) as this technology helps overcome the problems of source code pollution: The assertions are not inserted into the target system's source code, but into an aspect file, which is woven with the target system's bytecode. An initial case study has shown that the overhead due to instrumentation compares very well to previous approaches [8] and is likely to be acceptable in most situations, at least as far as testing is concerned. More empirical studies are however required.

Furthermore, we have shown how we dealt with aspects of the OCL specification that present serious instrumentation challenges (e.g. providing support for `@pre` and `oclIsNew()`) and reported on issues that we feel require future work (e.g. refinement of the OCL syntax and advanced optimization techniques).

References

1. Szyperski, C., *Component Software*. 2nd ed. 2002: ACM Press.
2. Briand, L.C., et al. *A Controlled Experiment on the Impact of the Object Constraint Language in UML-Based Development*. in *IEEE ICSM 2004*. 2004. Chicago, Illinois, USA.
3. Emmerich, W., *Engineering Distributed Objects*. 2000: Wiley.
4. Kleppe, A., J. Warmer, and W. Bast, *MDA Explained - The Model Driven Architecture: Practice and Promise*. 2003: Addison-Wesley.
5. Briand, L.C., Y. Labiche, and H. Sun, *Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code.* Software - Practice and Experience, 2003. **33**(7): p. 637-672.
6. Briand, L.C., W. Dzidek, and Y. Labiche, *Using Aspect-Oriented Programming to Instrument OCL Contracts in Java*. 2004. SCE-04-03. http://www.sce.carleton.ca/squall.
7. Finger, F., *Design and Implementation of a Modular OCL Compiler*. 2000, Dresden University of Technology.
8. Wiebicke, R., *Utility Support for Checking OCL Business Rules in Java Programs*. 2000, Dresden University of Technology.
9. LCI, *Object Constraint Language Environment (OCLE)*. http://lci.cs.ubbcluj.ro/ocle/.
10. Liskov, B., *Data Abstraction and Hierarchy.* SIGPLAN Notices, 1988. **23**(5).
11. Plösch, R., *Evaluation of Assertion Support for the Java Programming Language.* Journal Of Object Technology, 2002. **1**(3).
12. Lackner, M., A. Krall, and F. Puntigam, *Supporting Design by Contract in Java.* Journal Of Object Technology, 2002. **1**(3).
13. Appel, A.W., *Modern Compiler Implementation in Java*. 2nd ed. 2002: Cambridge University Press.
14. Elrad, T., R.E. Filman, and A. Bader, *Aspect-oriented programming: Introduction.* Communications of the ACM, 2001. **44**(10): p. 29-32.
15. AspectJ-Team. *The AspectJ Programming Guide*. [cited; Available from: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html.
16. Briand, L.C., W.J. Dzidek, and Y. Labiche. *Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging*. in *To appear in IEEE International Conference on Software Maintenance (ICSM 2005)*. 2005. Budapest, Hungary.
17. Warmer, J. and A. Kleppe, *The Object Constraint Language*. 1999: Addison-Wesley.
18. Bloch, J., *Effective Java: Programming Language Guide*. 2001: Addison Wesley.
19. Meyer, B., *Object-Oriented Software Construction*. 2nd ed. 1997: Prentice Hall.
20. Hussmann, H., F. Finger, and R. Wiebicke. *Using Previous Property Values in OCL Postconditions - An Implementation Perspective*. in *<<UML>>2000 Workshop - "UML 2.0 - The Future of the UML Constraint Language OCL"*. 2000. York, UK.
21. OMG, *Unified Modeling Language Specification 1.3*. 1999. http://www.omg.org.

Proposals for a Widespread Use of OCL

Dan Chiorean, Maria Bortes, Dyan Corutiu

Babes-Bolyai University, Computer Science Laboratory (LCI)
Str. M. Kogalniceanu 1, Cluj-Napoca 400084, Romania
chiorean@cs.ubbcluj.ro

Abstract. In spite of the fact that OCL and UML evolved simultaneously, the usage of the constraint language in modeling real-world applications has been insignificant compared to the usage of the graphical language. Presently, OCL is requested in new modeling approaches: Model Driven Architecture, Model Driven Development, Domain Specific Languages, Aspect Oriented Modeling, and various emerging technologies: Semantic Web, Business Rules. In this context, the question What has to be done for OCL to become the rule, not the exception, in the modeling domain? is more pressing than ever. The purpose of this paper is to propose an answer to this question, although not a complete one. Our work is an attempt to synchronize the language specification and its understanding, straight related to the language implementation in CASE tools, by proposing solutions for incomplete or non-deterministic OCL specifications. In order to manage the new extensions required for the constraint language, a new language structure is suggested.

Keywords: MOCL, UML, MOF, improving OCL, refactoring OCL , OCL extension

1 Introduction

The occasional use of OCL specifications in real-world applications is mainly due to the lack of appropriate tools supporting the language. This is quite a unanimous opinion in the expert community. In [1], John Daniels, one of OCL's grandparents, expresses his disappointment regarding OCL tools:

"Despite being a full part of UML 1.3 and being used extensively in the definition of UML itself, OCL is almost completely unsupported by current mainstream modeling tools, which is a shame because OCL allows us to be much more precise, especially when we specify component behavior."

Four years after the publication of Daniels' book, the state of facts changed, but not so much as needed. The position of one of the experts, published in [2], is:

"In recent years, the tool support for OCL has been pretty poor - as bad as the development environments we had to put up with in the eighties

(or even worse). I've struggled along with a couple of these less than usable tools to check that the OCL I've used in tutorials is correct, but quite often I end up fighting with the tool and give up."

The number of modeling directions requesting the use of OCL increases significantly by the day. In these circumstances, identifying the reasons of the unsatisfactory state of facts that persists in the OCL tool world and proposing reasonable solutions represents the first step. A clear, unequivocal and complete language specification is among the preconditions for conceiving and implementing the OCL tools required by real-world projects. The recommendations expressed in this paper are based on OCL features, on the language role and the requirements resulted from the application domain.

The paper is structured in 6 sections. Section 2 describes the characteristics that individualize this specification language. The OCL evolution is briefly analyzed in Section 3, where different unsolved language problems are highlighted. Some fallacies regarding the language are presented in the next section, Section 4. Finally, the authors' position regarding the improvement of the OCL specification is explained in Section 5 and the conclusions drawn are presented in the last section, Section 6.

2 Language Characteristics

Among the most important characteristics of OCL are: complementarity, conciseness and comprehensibility. OCL is not a stand-alone language. On one hand, OCL expressions need to be defined in the context of a model which provides the information for validation and evaluation of OCL specifications. On the other hand, the information provided by textual specifications complements the information provided by the graphical formalism, UML. For instance, in Figure 1 diagram the graphical formalism reflects the fact that a company has at least one employee and a person may be employed by zero or more companies:

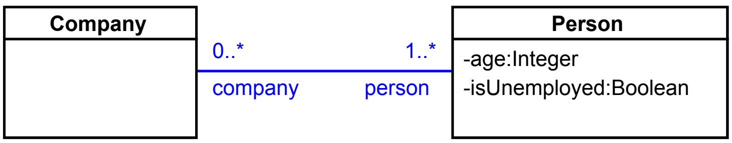


Fig. 1. A simple UML class diagram

However, a person cannot be hired by a company unless he/she is at least 18 years old. This fact cannot be graphically illustrated, but can be specified in an OCL invariant for the `Person` class:

```
context Person inv major:
    self.company->notEmpty() implies self.age > 18
```

In addition, the `isUnemployed` attribute of the `Person` class is a derived property since its value can be computed by navigating the company association end from the `Person` class. The expression which computes this value can be specified as follows:

```
context Person::isUnemployed: Boolean
derive:
    self.company->isEmpty()
```

As a straightforward consequence of complementarity, the OCL type system integrates with the type system of a modeling language (UML, MOF or other modeling languages). For instance, the UML primitive types: `Boolean`, `String`, `Real`, `Integer` map to the corresponding OCL primitive types and each type exported from a UML model (instance of a descendant of the `Classifier` metaclass) is a descendant of the `OclAny` type. The parallelism between the MOF based modeling languages' evolution and OCL's evolution is, from our point of view, another consequence of complementarity. New requirements emerged in the modeling domain generated new requirements for the textual formalism OCL. As an example, the usage of OCL for model transformations requires language extensions with constructs that allow transformation specification. The ongoing OMG standard, Query View Transformation [3], uses OCL extensively:

"All QVT packages are defined using EMOF from MOF 2.0, and extend the MOF 2.0 and OCL 2.0 specifications."

The transformation languages proposed in [4,5] are also based on OCL. The influence that Smalltalk exerts over OCL is reflected in the conciseness and comprehensibility of OCL specifications. A simple comparison among an operation specification in OCL, Smalltalk and Java is suggestive:

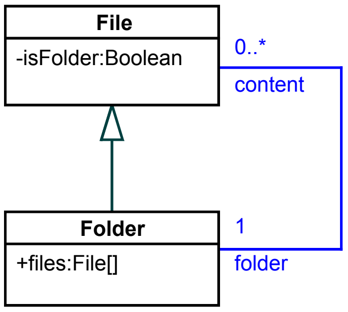


Fig. 2. The File and Folder model

```
-- OCL code
files():Set(File) = self.content->reject(f |f.isFolder)
```

```
"Smalltalk code"
files
    |folderFiles|
    folderFiles := self contents select [:f |f isFolder]
    .^folderFiles

// Java code
public Set files() {
    Set setContent = Folder.this.getContent();
    Set setReject = new HashSet();
    final Iterator iter = setContent.iterator();
    while (iter.hasNext()) {
        final File f = (File)iter.next();
        boolean bIsFolder = f.isFolder;
        if (!bIsFolder) setReject.add(f);
    }
    return setReject;
}
```

3 OCL Evolution

This section presents the language evolution and highlights aspects neglected in OCL specification and some unargued or controversial adopted decisions. Most of these aspects are analyzed in detail in Section 5, where some solutions are proposed.

The Object Constraint Language was conceived as a textual formalism complementary to the graphical formalism of UML. Obviously, the OCL evolution is aligned with the UML evolution.

The first official version of OCL, 1.1, was published in September 1997 [6], accompanying the 1.1 version of UML [7]. In that version, an OCL constraint had an implicit context: the model element, which the `Constraint` element was attached to. Conforming to the UML metamodel, the OCL constraint persistence could be achieved only through the instantiation of the `Constraint` metaclass. Beginning with the 1.3 version [8], the context and package concepts have been adopted. Their main purpose was to allow the specification of an explicit context for OCL constraints. A straightforward outcome of this adoption is that it allows constraints to be extracted from the model and managed (transferred and modified) by means of text files.

The 1.x OCL versions had a type system that comprised: primitive types, `OclAny`, `OclType`, `OclState` (since 1.3), `OclExpression` and `Collection` types. Nested collections were not allowed, thus successive navigations were followed by automatic collection flattening. Although since version 1.1 it has been stated that:

"OCL is a typed language, so each OCL expression has a type. In a correct OCL expression all types used must be type conformant",

the language specification did not fulfill this fundamental requirement. For instance, the type of the undefined value, `OclVoid`, was established later and adopted only in the 2.0 version [9]. Besides `OclVoid`, the type system of the 2.0 version was enriched with `OclTuple`, `OclMessage` and `OrderedSet`. `OclTuple` allows the description of database specific structures (e.g. result sets). `OclMessage` is used to monitor messages passed between objects, while `OrderedSet` is used to manage ordered sets. Also, in 2.0 version, nested collections are allowed, thus successive navigations are not followed by automatic collection flattening.

In order to provide an efficient and unambiguous support for the UML graphical formalism, OCL was extended with new language constructs. Thus, the navigation towards an `AssociationClass` in case of an auto-association was clarified in version 1.3, but the preferred syntactical solution overlaps with the one proposed for navigation of a qualified association with a single qualifier (see Section 5 Recommendations). The let construct was introduced in version 1.3 [8] and the def construct in 1.4 [10]. These constructs allow new observer functions to be defined in a `Classifier` context, along with granting their reusability. In the context of object-oriented architectures, the functions can be redefined in descendants. This important aspect was neglected in the OCL specification. Therefore, most OCL tools don't support redefinition of functions specified by means of the def construct.

Similar to most specification or programming languages, OCL evolved in order to provide better solutions or resolve open issues, as well as to allow adaptation to new requirements. Unfortunately, this evolution was based only to a small extent on requirements obtained from practical experience because this knowledge is hardly significant. Undoubtedly, the small number of available OCL tools and, mainly, their characteristics, represented one of the main reasons for this state of facts.

"The Amsterdam Manifesto on OCL" [11] offers a consistent set of solutions to a number of problems that remained unsolved in past versions. Along with [9], [11] provides the most exhaustive and consistent set of proposals regarding the evolution of OCL. Despite this fact, the authors explicitly state that not all the problems known at that time were approached:

> "During our meeting we discussed other subjects in the OCL standard that needed clarification. However, up to September 1999 we did not have the opportunity to discuss them further and include a clarification in this manifesto."

Some of the proposed solutions, for instance, the elimination of the generalization relation connecting `Collection` and `OclAny`, as well as the incorporation of the `oclIsNew()` operation in `OclAny`, are arguable, and we will try to prove this fact (see Section 5 - Recommendations).

OCL version 2.0 [9] offers a consistent set of solutions to a number of problems that remained unsolved in past versions. However, several problems mentioned as of [11] were still left unsolved or partially solved in OCL 2.0. This new version paid an increased interest to the language extension aspect as compared to the clarification of open issues inherited from previous versions. It is important to notice that the documentation for this version contains not only the Specification, but also a document [12] that describes unsolved problems and proposes several language extensions. In Section 5 - Recommendations we will look into into some of the problems that need to be solved for the language to fulfill all its promises.

4 Fallacies

Since the release of OCL's first version, several books [13,14,15,4,16] etc., have admirably promoted the language and the advantages it bestows. Also, many articles recommended the usage of OCL in a number of domains because of the promised advantages. Unfortunately, there are many misunderstandings disseminated not only by articles that attempt to present OCL in a poor light, but also by articles that try to promote OCL. Among OCL users, beginners or not, these false opinions may lead to misconceptions. Consequently, our recommendation to the OCL expert community is to analyze and shed light on these fallacies.

OCL is hard to understand and, as a consequence, difficult to use.
This statement is frequently publicized either without argumentation, or with a poor one that proves exactly the opposite [17]. A language's quality does not automatically grant the quality of the specifications realized in that language. The efficient use of OCL specifications requires, besides a good knowledge of the constraint language, the complete comprehension of the model for which the specifications were designed.

OCL has to be side effect free.
This characteristic originates from the first versions. At that time, the purpose of the language was limited to assertion and observer operation specifications. In these cases, the evaluation of OCL specifications should not alter the model state. Lately, the usage of OCL in transformation specification and in behavior description for non-query operations requires imperative constructs [4,5]. The side effect free property is not a quality, but a requirement for assertion and observer operation specification. New objectives like those mentioned above need an imperative language.

The OCL metamodel must be used for OCL expression persistence,
similar to the way the UML metamodel is used to ensure UML model interchange through the XMI format. Such an approach is highly inefficient. A textual format is the most appropriate interchange format for textual formalisms, as it was already proven by programming languages. Also, this approach involves minimal resource consumption (memory, processor time) and offers obvious advantages in specification management.

OCL Collections must be homogenous.
It is not allowed for a collection to contain elements whose single common an-

cestor is `OclAny`. This restriction is not reasonable. Object-oriented programming languages provide means to manage heterogenous collections through a common base interface for collection elements (e.g. `Object` in Java). In OCL's case, the role of this common base interface can be played by `OclAny`. The existence of a root interface for all types also solves the type resolution problem that may appear while analyzing an `if-then-else-endif` construct.

5 Recommendations

In order to be achieved, the objective stated at the beginning of the paper - the widespread use of OCL specifications for realizing real-world applications by means of modern modeling technologies - requires a much more careful requirement analysis both for the constraint language and for the CASE tools that support its usage.

5.1 Language Requirements

Lately, the modeling domain has evolved considerably, demanding new features for the textual language that accompanies the graphical language. Among these new features there are: support for metamodeling (MOF based and Domain Specific Modeling), model transformation, Aspect Oriented Modeling, extensions for behavior specifications, support for model testing and simulation, ontology development and validation for the Semantic Web, etc.

The extension of OCL with new features should not affect its simplicity, coherence and clearness. In this context, the extension of OCL requires first a language refactoring by means of: modularization, elimination of redundant constructs or those having model dependent semantics, unequivocal type resolution for all OCL expressions, clarification of open issues related to the type system, complete and unambiguous specification of evaluation results for all OCL expressions.

5.1.1 Language modularization By modularization we understand the organization of language concepts in highly cohesive and lowly coupled units - language modules. These modules describe all aspects related to the contained concepts, including their syntax and semantics.

The main purpose of modularization is to obtain an Open-Closed architecture [18]. Such an architecture offers advantages both for language users and for language developers: language users are given the opportunity to concentrate only on language modules that are of interest for their application domain, while language developers have to concentrate only on those modules that are subject of creation or modification, being assured that their modifications would not be propagated across other language modules.

The OCL version we considered for modularization is 2.0, because it takes into account the use of OCL in conjunction with the MOF modeling language and contains the richest set of concepts.

For each application domain, OCL modules can be developed to cover domain specific concepts. These modules may be assembled with other OCL modules in a dialect associated to the modeling language.

In a MOF-based environment the model defined types integrate with the OCL type system. Model navigation is the basic functionality without which no textual specification would be possible. Consequently, we propose to group together, in a language module named "Core OCL", the OCL type system and the language concepts that realize model navigation. This module is independent of any other module and it is used (directly or indirectly) in any other module. "Core OCL" contains only the language concepts needed to navigate a MOF model. For modeling languages with concepts that require specific navigations (e.g. qualified associations in UML), "Core OCL" can be extended with specific language concepts that realize these navigations.

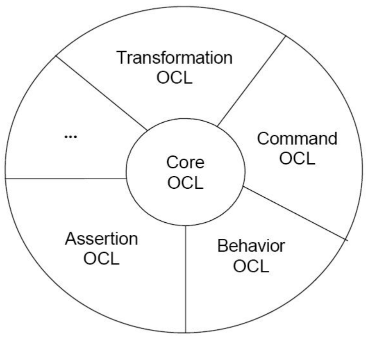


Fig. 3. A modular architecture for OCL

OCL 2.0 allows constraint specification (invariants, pre and post-conditions, guards), behavior specification for query operation (body and def constructs), initial and derived property values (init and derive constructs). The above-mentioned constructs can be separated in two modules: "Assertion OCL" which contains constructs for constraint specification and "Behavior OCL" which contains constructs for behavior specification and for initial and derived property values. The latter module could also contain mechanisms for the management of exceptions, an aspect completely absent from the OCL specification.

The complete behavior specification and model transformation [5] - a fundamental mechanism in MDA - would not be possible without adding modifiers[1] to OCL - constructs that produce side effects. Moreover, these constructs are also needed for a "Command OCL" module which allows the construction

[1] modifier = operation that alters the state of an object

and modification of object snapshots, similar to the USE command language [19]. This module is useful for model testing and simulation. As more modules depend on "modifiers", these constructs can be isolated in a separate module, "Imperative OCL". Obviously, the modules "Behavior OCL", "Command OCL" and "Transformation OCL" (a module for transformation specification) depend on it.

5.1.2 Elimination of redundant constructs or those having model dependent semantics

5.1.2.1 Redundant constructs. In order to preserve language conciseness and comprehensibility, each language feature should be realized through one language construct; more constructs for the same language feature do not enrich the language, but increase its complexity. A redundancy example is the `oclIsNew()` operation that belongs to the `OclAny` type. This operation can be used exclusively in postconditions and its role is to distinguish objects that have been created during the execution of the operation. For example, the execution of the `hire(p:Person)` operation from the `Company` class (see Figure 3) implies the instantiation of the `Job` class. In order to check whether, right after operation execution, the person's wage is higher than the minimum wage, the following postcondition can be specified:

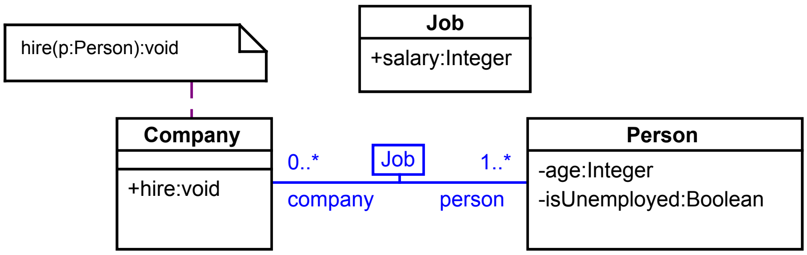


Fig. 4. UML class diagram

```
context Company::hire(p: Person)
post:
    p.job->select(j |j.oclIsNew())->forAll(j | j.salary > 800)
-- or,
post:
    p.job->excludesAll(p.job@pre)->forAll(j | j.salary > 800)
```

The above example contains two postconditions that express differently the same constraint: the first one uses the `oclIsNew()` operation while the second uses the `@pre` construct that holds the set of jobs for Person p before the execution of `hire(p:Person)` operation.

The `oclIsNew()` operation can be completely substituted by the `@pre` construct, because the newly created objects during operation execution can be identified by removing from the collection of existing objects, at the end of operation execution, the collection of existing objects prior the operation execution, obtained with the `@pre` construct.

Along with its redundancy, `oclIsNew()` operation brings in some other problems. Although it is part of `OclAny` interface, it cannot be called on any OclAny descendant. For instance, `oclIsNew()` has no sense for the primitive types: `Boolean`, `String`, `Real` and `Integer`. The same problem occurs for the `allInstances()` operation in `OclType`. `oclIsNew()` also exhibits an unusual behavior: it is the only operation in OCL standard library that can be used exclusively in postconditions, in spite of the fact it belongs to the `OclAny` interface.

The operation `any()` can be considered another redundant construct specified on collections. The last OCL specification, [9] defines `any()` as being realized with the help of `asSequence()` and `first()`, operations defined on `Collection` and `Sequence`. The operation `any()` can be also replaced with `sortedBy()` succeeded by `at(i)`, where `0<=i<=size`.

5.1.2.2 Evaluation of non-deterministic constructs in OCL. Another controversial topic regarding OCL specifications, topic debated on the pUML mailing list, is represented by non-deterministic constructs. Recently, Mr. Thomas Baar published a detailed paper on this topic [20], where, among other things, it is stated: "specifications in constructive languages using non-deterministic constructs can easily be rewritten in OCL without using non-deterministic constructs". This conclusion coincides with our proposal regarding the removal of the `any()` operation. In fact, as specialists agree, the `asSequence()` operation is also non-deterministic. This unpleasant situation can be removed by stating that in the obtained Sequence, the order of elements remains unchanged from the initial collection.

On the pUML Mailing list, the `sortedBy()` operation is also mentioned as a possible cause for non-deterministic behavior. In our opinion, this aspect can be easily removed by mentioning that if the strict ordering criterion is not accomplished, the order of compared elements remains unchanged. For example, the result of evaluating the specification `Setp1, p2->sortedBy(e | e.name)`, where `p1.name = p2.name`, will be `Sequencep1, p2` and not `Sequencep2, p1`, because the initial order remains unchanged.

The above-mentioned additional information is important for CASE tools implementors because it offers the opportunity to have different CASE tools that produce the same results for identical inputs.

5.1.2.3 Constructs with model dependent semantics. The semantics of the following constraint is model dependent:

```
context A
inv:
    self.b[c].notEmpty()
```

For the model presented in Figure 5, `self.b[c]` represents the result of navigating the auto-association class `B` from class `A`, through the association end named `c`.

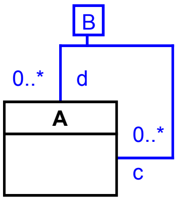


Fig. 5. First model context: navigation of an auto-association class

For the model presented in Figure 6, `self.b[c]` represents the result of navigating the qualified association between `A` and `B` through the `B` class, having the value of the `c` attribute of `A` class as value for the `id` qualifier.

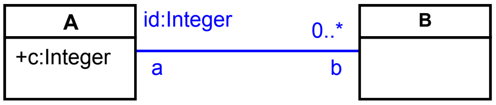


Fig. 6. Second model context: navigation of a qualified association

The above example illustrates how the same OCL expression can have multiple semantics, depending on its model context. This ambiguity can be eliminated through designating a new language construct for navigation of the auto-association classes. Since this solution keeps the OCL expression context independent, it helps both the user (in unequivocally understanding the OCL specification) and the OCL compiler providers (in detecting compilation errors in early stages, without complex context analysis).

The concrete syntax we propose for navigating auto-association classes is the following:

```
context A inv:
        self.b#c.isEmpty()
```

where the above constraint applies to the model in Figure 5. By replacing the square brackets with '#' we can unambiguously identify this situation.

5.1.3 Improving type checking for OCL expressions In case of `if-then-else-endif` expressions:

```
if booleanExpression (condition)
        then thenExpression
        else elseExpression
endif,
```

the only information about the `thenExpression` and `elseExpression` is that they are `OclExpressions` [9] (page 46). As presented in [9] (page 128, Figure 28), the `Collection` type is not a descendant of `OclAny`. In this context, if the type of `thenExpression` is a Collection type and the type of `elseExpression` is not a Collection type, the type of the if expression cannot be determined. Consequently, the if expression is rejected by a OCL compiler because the `thenExpression` and `elseExpression` do not have a common base type. The above-mentioned problem has at least two reasonable solutions:

- the acceptance of `Collection` type as a descendent of `OclAny`,
- the definition of a constraint that ensures the existence of a common base type for `thenExpression` and `elseExpression`.

The first solution is also beneficial for a natural specification of nested collections.

5.1.4 A better approach for the type cast operation Concerning the type cast operation, widely used in OCL specifications, in [9] it is stated:

> "When it is certain that the actual type of the object is the subtype, the object can be re-typed using the operation `oclAsType(OclType)`. This operation results in the same object, but the known type is the argument `OclType`. When there is an object object of type `Type1` and `Type2` is another type, it is allowed to write: `object.oclAsType(Type2)` — evaluates to object with type `Type2`. An object can only be re-typed to one of its subtypes; therefore, in the example, `Type2` must be a subtype of `Type1`. If the actual type of the object is not a subtype of the type to which it is re-typed, the expression is undefined (see ("Undefined Values"))."

As stated in [9] about "Accessing overridden properties of supertypes" (page 20, paragraph 7.5.8), the casting of a type to a supertype is also allowed. In case the source and target types of the cast operation are not related, the compiler should report an error.

5.1.5 Detailed specification for evaluation of expressions containing undefined values The evaluation of operations invoked on collections that contain undefined values is superficially approached in all OCL specifications. The fact that the collection contains undefined values should not imply that all collection operations invoked on that collection result in undefined. For instance:

- the evaluation of the collection size should always return a positive integer value:

```
Bag{1, 9, undefined, undefined }->size = 4
Sequence{1, 9, undefined, undefined }->size = 4
```

– the evaluation of the exist() operation should iterate the entire collection and not terminate the iteration when an undefined value is encountered:

```
Collection{undefined, 1, 9, undefined}->
                exist(e |e > 7) = true
Collection{1, 9, undefined}->
                exist(e |e = 7) = undefined
```

– the evaluation of the `any()` operation on a collection that contains at least one element that satisfies the condition of the `any()` operation should result in one of these elements:

```
Collection{1, 9, undefined, undefined}
                ->any(e | e>1) = 9
Collection{1, 9, undefined}
                ->any(e | e>10) = undefined
```

– the evaluation of a `forAll()` operation on a collection that contains one element that does not satisfy the `forAll()` condition should result in false:

```
Collection{1, 9, undefined}
                ->forAll(e | e>2) = false
Collection{1, 9, undefined}
                ->forAll(e | e>=1) = undefined
```

The use of the undefined value is helpful in specifying operation behavior. Therefore, a new kind of Expression, UndefinedLiteralExpression, needs to be defined in the OCL grammar.

5.2 Tool Requirements

Due to the complementarity between the textual and graphical formalism, there are two directions in OCL tool development:

– *stand-alone tools* - they offer support for both formalisms, implementing features specific to CASE tools along with features characteristic to the textual formalism; these tools are based on proprietary repositories and graphical editors;
– *plug-ins for CASE tools* - they offer support only for the textual formalism and use the CASE tools' repositories and graphical editors (e.g. Octopus and Pampero are OCL plug-ins for EMF-Eclipse, and ModelRun is a Rational Rose plug-in).

For a broad acceptance, OCL tools must prove their efficacy and straightforwardness. Starting from the proposed modular architecture of the OCL language, a variety of OCL tools may be conceived and implemented, ranging from general tools, that implement most of the language modules, to more specific tools, that implement only modules that target a well defined functionality, tools for model checking against a set of OCL rules [21], tools for model transformation, tools for model simulation, etc.

The fundamental functionalities of any OCL tool are: compilation, evaluation and debugging of OCL specifications. Along with the OCL support, these tools may provide additional functionalities as code generation from OCL specifications [21] to various programming languages or notations.

Features like user friendly IDE functionalities: syntax highlighting, auto-indentation, auto-completion, interactive debugging, help system are also important factors that influence the widespread use of such tools.

If OCL tools fail to reasonably provide the above-mentioned functionalities, it will be almost impossible to use the OCL language at its true value.

6 Conclusion

Our belief is that, for the success of the newly emerged modeling technologies - MDA, MDD, MDE, LDD, DSM - the usage of a textual formalism as OCL represents more than a UML routine, it is a must.

The textual formalism has real perspectives of a broad use. In order to materialize these perspectives, the language and the tools supporting it have to adapt to new requirements while preserving the spirit of the language and its advantages.

It is mandatory to take into consideration the lessons learned throughout the OCL evolution. The development and adoption of a new version of the OCL standard should be primarily guided by technical arguments and validated through a productive debate, not by a vote majority built around individual interests. Also, the illustration of several misconceptions about the constraint language, followed by a clear argumentation of the state of facts, is useful both for decision makers and users alike.

References

1. John Cheesman and John Daniels. *UML Components – A Simple Process for Specifying Components-Based Software*. Addison Wesley, 2001.
2. UML and Agile Development discussion board. OCL evaluator. Available at http://parlezuml.com/blog/?postid=13, April 2005.
3. OMG QVT Merge Group. Revised submission for MOF 2.0 - query/views/transformations rfp, v. 1.6. OMG Document formal/04-08-16, August 2004.
4. Tony Clark and Jos Warmer. *Object Modeling with the OCL – The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.
5. The ATLAS group. *Atlas Transformation Model, User Manual, version 0.5*. Available at http://www.sciences.univ-nantes.fr/lina/atl/atlProject/atlas/, 2005.
6. OMG. Object constraint language specification v. 1.1. OMG Document formal/97-08-08, August 1997.
7. OMG. Unified modeling language specification v. 1.1. OMG Document formal/97-08-03, August 1997.

8. OMG. Unified modeling language specification v. 1.3. OMG Document formal/99-06-09, June 1999.
9. OMG. Object constraint language specification v. 2.0. OMG Document formal/03-10-14, October 2003.
10. OMG. Unified modeling language specification v. 1.4 with action semantics. OMG Document formal/02-01-09, January 2002.
11. Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. UML 2.0 request for information response - the amsterdam manifesto on OCL. Available at http://www.trireme.com/whitepapers/design/components/oclmanifesto.pdf, 1999.
12. OMG. Ocl 2.0 final rtf/ftf report. OMG Document formal/05-06-05, June 2005.
13. Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison Wesley, first edition edition, 1999.
14. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, second edition edition, 2003.
15. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained, the Model Driven Architecture: Practise and Promise*. Addison Wesley.
16. Desmond D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML - The Catalysis Approach*. Addison Wesley, 1999.
17. Dan Chiorean, Maria Bortes, and Dyan Corutiu. Good practices for creating correct, clear and efficient OCL specifications. In *Proceedings of 2nd Nordic Workshop on the Unified Modeling Languages*, pages 127–142, http://crest.cs.abo.fi/nwuml04, 2004.
18. Robert C. Martin. The open-closed principle. *C++ Report*, 1996.
19. University of Bremen. The USE tool. Available at http://www.db.informatik.uni-bremen.de/projects/USE.
20. Thomas Baar. Non-deterministic constructs in OCL – what does any() mean. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *Proc. 12th SDL Forum, Grimstad, Norway, June 2005*, volume 3530 of *LNCS*, pages 32–46. Springer, 2005.
21. Dan Chiorean, Maria Bortes, Dyan Corutiu, and Radu Sparleanu. UML/OCL tools - objectives, requirements, state of the art - the OCLE experience. In *Proceedings of the NWPER'2004*, pages 163–180, http://crest.cs.abo.fi/nwper04, 2004.

OCL and Graph-Transformations – A Symbiotic Alliance to Alleviate the Frame Problem[*]

Thomas Baar

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
thomas.baar@epfl.ch

Abstract. Many popular methodologies are influenced by Design by Contract. They recommend to specify the intended behavior of operations in an early phase of the software development life cycle. In practice, software developers use most often natural language to describe how the state of the system is supposed to change when the operation is executed. Formal contract specification languages are still rarely used because their semantics often mismatch the needs of software developers. Restrictive specification languages usually suffer from the "frame problem": It is hard to express which parts of the system state should remain unaffected when the specified operation is executed. Constructive specification languages, instead, suffer from the tendency to make specifications deterministic.
This paper investigates how a combination of OCL and graph transformations can overcome the frame problem and can make constructive specifications less deterministic. Our new contract specification language is considerably more expressive than both pure OCL and pure graph transformations.

Keywords: Design by Contract, Behavior Specification, Graph Grammars, OCL, QVT

1 Motivation

Design by Contract (DbC) [1, 2] encourages software developers to specify the behavior of class operations in an early phase of the software development life cycle. Precise descriptions of the intended behavior of operations can be of great help to grasp design decisions and to understand the responsibilities of classes identified in the design.

The specification of behavior is given in form of a *contract* consisting of a pre- and a post-condition, which clarify two things: The pre-condition explicates all conditions that are expected to hold whenever the operation is invoked. The post-condition describes how the system state looks like upon termination of the operation's execution. Basically, contracts can be formulated in an informal way

[*] This work was supported by HASLER-Foundation, project DICS-1850.

or using a formal language such as OCL. Formally specified contracts have the advantage to be a non-ambiguous criterium for the correctness of a given implementation. Furthermore, contracts written in a formal language are machine readable and can be automatically processed in later stages of the software development life cycle, e.g. for the purpose of test case generation [3].

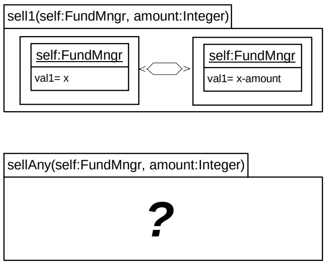
There are many specification languages available to define contracts formally. Despite their differences at the surface level, all languages can be divided into only two classes. The classification is based on the technique to specify the post-condition of a contract. *Restrictive specification languages* formulate the post-condition in form of a predicate, i.e. a Boolean expression, which restricts the allowed values for properties in the post-state. Well-known examples for restrictive languages are OCL, JML, Z, and Eiffel. *Constructive specification languages* interpret post-conditions not as restrictions on the post-state but – conceptually completely different – as updates, which transform the pre-state into the post-state. Well-known examples for constructive languages are B, ASM, graph transformations, and UML's Action Language.

The main disadvantage of using restrictive languages is the well-known frame problem [4]: The predicate for the post-condition can hardly express which parts of the system should not change.

```
context FundMngr::sell1(amount)
    post: val1 = val1@pre-amount


context FundMngr::sellAny(amount)
    post: val1+val2 = val1@pre+
                      val2@pre-amount
```

(a) Restrictive specifications using OCL


sell1(self:FundMngr, amount:Integer)
self:FundMngr
val1= x
self:FundMngr
val1= x-amount
sellAny(self:FundMngr, amount:Integer)
?

(b) Constructive specifications using graph transformations

Fig. 1. Specification of 'simple' operations

Suppose, a (simplified) class `FundMngr` has two attributes `val1`, `val2` representing the value of two stock depots. The intended behavior of operation `sell1(amount)` is to sell shares of value `amount` from the first depot. Typically, the operation `sell1()` would be specified in OCL as shown in the upper part of Fig. 1 (a). This specification, however, does not capture the intended semantics because also implementations of `sell1()` conform to the OCL specification that not only decrease `val1` by `amount` but in addition change the value of `val2`.

Constructive specification languages do not suffer from the frame problem but from a severe, complementary problem. Since a constructive specification describes how to 'construct' the post-state out of the pre-state, it prescribes the implementation of the operation completely. Consequently, all decisions on the operation's behavior have to be taken in time of writing the specification and cannot be deferred to the implementation phase. Hence, the constructive specification and the implementation of an operation coincide.

The pros and cons of constructive specification languages are illustrated in Fig. 1 (b). Here, the behavior specification is given in form of a graph transformation rule, which consists of two graph patterns called left-hand side (LHS) and right-hand side (RHS). They define *how* to 'construct' a post-state out of a given pre-state: The pre-state is assumed to be represented as an object diagram. In a first step, all subgraphs of the object diagram are searched that matches with LHS. In a second step, each matching subgraph is rewritten with a new subgraph that can be uniquely computed based on RHS (see Sect. 3.2 for details).

The specification of operation `sell1()` in Fig. 1 (b) is read as follows. Whenever in the pre-state a subgraph can be found consisting of object `self` whose value for attribute `val1` matches with a (fresh) variable `x` then this subgraph is rewritten by the same object `self` whose attribute `val1` has now the value `x-amount`. Note that object `self` is passed as a parameter to the rule which lets LHS match with only one subgraph of the pre-state. All objects and links of the pre-state that are not part of the matching subgraph remain unchanged. The same holds for the values of all attributes of object `self` that are not mentioned in RHS. Consequently, if an implementation of `sell1()` would change for object `self` the value of attribute `val2` then this implementation would not conform to the constructive specification.

In order to illustrate the disadvantages of constructive specification languages we consider a second operation `sellAny(amount)` whose intended behavior is to sell shares of value `amount` but it is not important whether shares from the first or from the second depot are sold. The final implementation of `sellAny()`, of course, had to realize an algorithm that determines for each depot the number of shares to be sold, but the decision, which algorithm should be taken, is intentionally deferred to the implementation phase.

A contract for `sellAny()` can easily be given using a restrictive language. Figure 1 (a) shows an OCL contract where the post-state is *underspecified*: if a concrete pre-state is given, the post-state properties `val1`, `val2` can have more than one solution. In other words, the post-state is not (always) determined by the pre-state and the contract. We call such contracts *non-deterministic*. Non-deterministic contracts cannot be expressed by purely constructive languages (see Fig. 1 (b)) because there is no unique update that could by applied to the pre-state in order to construct the post-state (if there were such an update, the contract would be deterministic).

This paper investigates how the expressive power of constructive languages – as an example we consider graph transformations – can be improved to master non-deterministic contracts. In Sect. 3, graph transformations are extended with restrictive specification elements (OCL clauses). In its extended version, graph transformations are more powerful but still not powerful enough to formalize all contracts that are relevant in practice. Thus, a second extension is discussed in Sect. 4, which allows to simulate the loose semantics of restrictive languages. To summarize, the proposed extensions of graph transformations enable software developers to write formal contracts that (1) do not suffer from the frame problem, (2) are non-deterministic, and (3) allow to change a state freely.

Related work. The idea to use graph transformations to formalize contracts is not novel. There are even already tools for this purpose available [5, 6]. The examples we found in the literature, however, are always deterministic contracts, which do not require to extend graph transformations with restrictive specification elements.

The idea to extend graph transformations with OCL clauses has been adopted from the Query/Views/Transformations proposal (QVT) [7], which is a response on a corresponding request for proposals by the OMG. In Sect. 3, the QVT approach is, however, put into a broader context by providing the link from model transformation (the original application domain of QVT) to formal contract specification.

Another attempt in the literature to make graph transformations less deterministic is by Heckel et al. in [8]. Having the same goal as our approach of Sect. 4, they first introduce graph transformations based on a loose semantics and make this notation, in a second step, more constructive by specifying explicit frame conditions on selected types.

Combining OCL with object diagrams has been explored in the literature also for a different target than contract formalization. The language VOCL (Visual OCL) uses collaborations to represent OCL constraints in a visual format for better readability [9]. Similarly, the proposal made by Schürr in [10] is inspired by Spider diagrams and aims at a more readable, graphical depiction of OCL constraints. The approaches described in [9, 10] cannot be compared with the approach presented in this paper because they have a fundamentally different goal. Firstly, [9, 10] do not use OCL in order to improve the expressive power of a graphical formalism. Instead, the graphical formalism is merely used as an alternative to OCL's textual standard syntax. Secondly, our approach targets only operation contracts whereas [9, 10] aim at a visualization of any kind of OCL constraints including invariants.

2 Restrictive Languages and the Frame Problem

In this section, we analyze why restrictive languages can hardly avoid the frame problem. The frame problem is much more complex than the trivial example in Sect. 1 was able to illustrate. This complexity makes naive approaches to tackle the frame problem, as for instance by adding frame axioms to the post-condition, very questionable. Some restrictive languages try to alleviate the frame problem by inventing a new clause for contracts. The new clause describes which parts of the system state must remain unchanged when executing the operation.

2.1 Example: CD-player

A formal specification of that are provided by CD-players will illustrate well the complexity of the frame problem. In Section 3, this example will be used again to point out limitations of constructive languages.

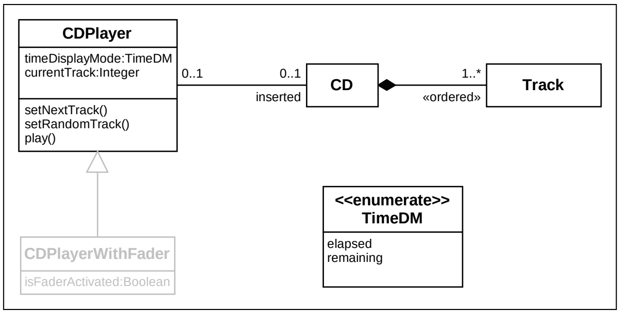


Fig. 2. Static model of CD-player scenario

The main purpose of CD-players is to entertain people and to play the content of compact discs (CDs). The content of a CD is organized by tracks that are burned in a certain order on the CD. We want to assume that a CD can be played in two modes. In the normal mode, all tracks on the CD are played in the same order as they appear on the CD. In addition, the CD-player can work in a shuffle mode in which the tracks are played in a randomized order. Finally, we want to assume that a CD-player has a display on which, depending on the chosen display mode, the elapsed or remaining time for the current track is shown.

This CD-player scenario is modeled straightforwardly by the class diagram shown in Fig. 2. The subclass `CDPlayerWithFader` can be ignored for the moment; later we will come back to it when discussing how object-oriented designs can evolve (e.g. by adding new subclasses) and which consequences this has on the semantics of operation contracts.

In the next subsection, we will focus on the formal behavior specification for the operations `setNextTrack()` whose intended semantics is to determine the next track to be played if the CD-player is working in the normal mode. The operation `setRandomTrack()` will be specified in Sect. 3 and determines the next track if the CD-player works in the shuffle mode.

2.2 Complexity of the Frame Problem

The intended semantics of operation `setNextTrack()` is to move one track forward on the CD and to increase the value of attribute `currentTrack` by one. The formalization of this behavior in a restrictive language such as OCL seems to be straightforward but there are some traps one can fall into.

```
context CDPlayer::setNextTrack()
    pre: self.inserted->notEmpty()
    post: self.currentTrack = (self.currentTrack@pre mod
    self.inserted.track->size()) + 1
```

This contract has some merits since it resolves ambiguities that were hidden in the informal description of the behavior. The first important information is expressed by the pre-condition saying that the CD-player assumes to have a CD inserted whenever the operation `setNextTrack()` is invoked. Note that this assumption is indeed necessary because the post-condition navigates over the currently inserted CD. The second merit of the contract is to make explicit the behavior of `setNextTrack()` when the current track is the last one on the CD. Reasonable variants might be to set `currentTrack` to zero (and thus to stop playing) or to continue with the first track on the CD as it is stipulated by our OCL constraint.

Although the OCL contract clarifies the informally given specification in some respects, it does not capture completely the intended behavior. According to the formal semantics of OCL in [11], an implementation still fulfills the contract even if it would not only change the value of `currentTrack` but also the display mode (attribute `timeDisplayMode`). Or the implementation could create/delete other objects, or could change the state of other objects, or could change the connections (links) between objects.

2.3 Strategies to Overcome the Frame Problem

A very naive strategy to exclude unintended implementations is by adding equations (so-call *frame axioms*) to the post-condition in order to make explicit which parts of the state should remain unchanged. In case of `setNextTrack()`, one had to add equations such as `self.timeDisplayMode=self.timeDisplayMode@pre` and `CD.allInstances=CD.allInstances@pre` and .... The huge number of necessary equations, however, let the size of the post-condition explode. Another drawback of this 'solution' is the need to rewrite all contracts of the design whenever the state space of the designed system is changed, e.g. by introducing a new class `CDPlayerWithFader`.

Unfortunately, this poor strategy of adding frame axioms is currently the only possibility, how OCL users can try to tackle the frame problem. To our knowledge, there has not been any attempt yet to make the language OCL more expressive so that users can easily add to a contract some information on which parts of the system remain unchanged.

Other restrictive languages have tried to tackle the frame problem by adding syntactical constructs which makes the semantics of a contract stronger. Users of the specification language Z [12] can separate the state space of the system into one part that is not affected by the operation and one part that can change freely as long as the restrictions formulated in the post-conditions are satisfied. The language JML [13], a contract specification language for methods implemented in Java, offers besides pre-/post-conditions an additional clause *assignable* (also known as *modifies*) where all locations that might change their value must occur.

There has been also attempts in the literature to 'compute' then changing part of the system merely based on the post-condition [14]. This, however, makes formal reasoning on formal specifications much more complicated.

3 Constructive Languages and Non-Deterministic Contracts

After the last section has pointed out the most important drawback of restrictive languages, this section discusses a corresponding problem of constructive languages, namely, the principal obstacles for keeping the operation behavior to a certain degree unspecified. This can be only achieved by non-deterministic contracts.

Graph transformations are introduced as a constructive specification language. It is discussed, why pure graph transformations can specify the operation `setNextTrack()` but fail to specify `setRandomTrack()` correctly. To overcome this problem, we finally discuss a combination of constructive and restrictive specification style.

3.1 Non-deterministic Contracts

Non-deterministic contracts are necessary when not all details of the operation behavior should be fixed in time of writing the contract.

The intended behavior of `setRandomTrack()` is a typical example for a non-deterministic contract. The operation name set*Random*Track might be misleading as it might set up the expectation that our contract will enforce a true randomized behavior of the implementation in the sense that invoking the operation twice in the same state can result in different post-states. Note that this kind of randomness cannot be expressed by a contract (neither in OCL nor in any other contract language) because it would require to describe formally the behavior of multiple invocations whereas a contract can specify only the behavior of a single invocation.

The specification of `setRandomTrack()` in OCL looks as follows:

```
context CDPlayer::setRandomTrack()
   pre: self.inserted->notEmpty()
   post: Set{1..self.inserted.track->size()}
            ->includes(self.currentTrack)
```

This contract suffers again from the frame problem but, if this is ignored for a while, the post-condition keeps intentionally the exact post-state open and thus allows many different implementations. Even, an implementation that constantly sets attribute `currentTrack` to 1 was possible and would conform to this contract.

3.2 Graph Transformations as a Constructive Language

Graph transformations have their roots in graph grammars and were originally applied to describe the syntax of graphical languages. A graph grammar is a set of rules that specifies all syntactically correct sentences of a visual language. A visual sentence is syntactically correct if it can be derived by the recursive application of grammar rules starting on an initial graph. Graph grammars mimic in many respects the traditional syntax definition of textual languages by EBNF rules. Instead of sequences of strings, a graph grammar generates sets of visual objects placed in an n-dimensional space, or – to describe the outcome more abstract – a graph grammar generates (typed) graphs. From a more abstract point of view, rule applications are nothing but graph transformations and graph grammar rules are an elegant way to specify these graph transformations.

It has also been recognized in the literature (see [6] for a survey and [5] for a concrete example) that graph transformations can be used to specify the behavior of operations. System states can easily be represented as graphs, e.g. in form of object diagrams, and system state changes can be encoded as a transformation of graphs.

A graph transformation rule consists of two graph patterns called left-hand side (LHS) and right-hand side (RHS). Graph patterns are normal graphs whose elements, i.e. nodes and links connecting some nodes, are identified by labels. It is possible to use both in LHS and RHS the same label for the same kind of elements (nodes or links). The application of a graph transformation rule on a given graph is roughly described in two steps. In the first step, it is checked whether the given graph has a subgraph that matches with LHS. If not, the rule is not applicable on this graph. If yes, the matching subgraph is substituted by a new graph derived from RHS under the matching obtained in step 1. If a label for an element occurs only in LHS but not in RHS then the matching element is removed, if it occurs in RHS but not in LHS then a new element is created, if a label occurs in both LHS and RHS then the element is remained unchanged during the application of the rule.

Besides this basic version of graph transformation rules, where LHS and RHS consist of simple nodes and links, modern graph transformation systems offer much more sophisticated elements to describe patterns such as typed nodes, multiobjects, negative application conditions (NACs), parameters, etc. In the rest of the paper, we will use the graph transformation system QVT submitted as a proposal to the OMG for the standardization of model transformations. For details on the syntax/semantics of this formalism, the interested reader is referred to [7]. A bigger example on how QVT can be used as a contract specification language is given in [15].

As a simple example for a behavioral specification using graph transformations, Fig. 3 shows a rule specifying the intended behavior of `setNextTrack()`.

The graph patterns LHS, RHS use typed nodes (e.g. `self:CDPlayer`) that must comply to the system description given in Fig. 2. The LHS of the rules serves two things. First, it imposes restrictions that must hold in order to make

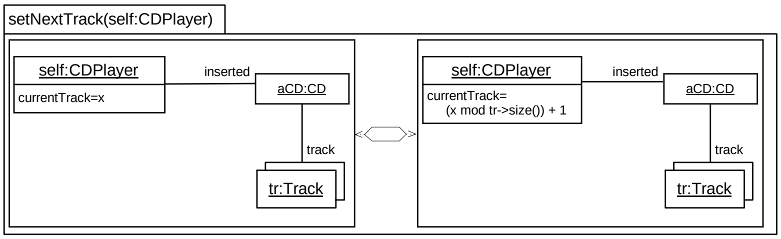


Fig. 3. Specification of `setNextTrack` with QVT

the rule applicable for the given state. For `setNextTrack()`, the effective restriction is that the CD-player `self` has a CD inserted (expressed by the link between `self` and `aCD`). The second purpose of LHS is to query the pre-state and to extract information that is important for the post-condition encoded by RHS. In our example, the variable `x` extracts the current value of attribute `currentTrack` and multiobject `tr` denotes the set of all tracks of the inserted CD. Note that the attribute `currentTrack` and the multiobject `tr` could have been omitted in LHS and the rule would still be applicable on exactly the same set of graphs as before.

The RHS of `setNextTrack()` is almost identical to LHS except for the value of attribute `currentTrack`. Consequently, applying the rule on a state will change only the value of `currentTrack` on the object `self` and nothing else. The new value of this attribute is computed based on the information queried during the first step of the rule application.

3.3 Mixing Constructive and Restrictive Languages

Graph transformation rules, as they were explained so far, can capture deterministic contracts in an elegant way whereas it seems hopeless to use them for non-deterministic contracts.

Fortunately, there is a solution and the same problem has been already tackled by other constructive languages. The language B, for example, offers, besides a pseudo-programming language for computing the post-state, the construct ANY-WHERE. This construct causes a non-deterministic split in the control flow and connects the same pre-state with possibly many post-states. The non-deterministic choices are, however, restricted by a predicate, which has to be evaluated in all control flows to true. In other words, constructive and restrictive specification style is mixed. The formal semantics of ANY-WHERE is defined in [16]. For an example-driven explanation of ANY-WHERE, the reader is referred to [17].

By integrating ANY-WHERE, the language B has lost its purely constructive semantics. The gain of expressive power is paid by loosing the executability of B specifications. This makes tool support for B more challenging but not impossible [18, 19].

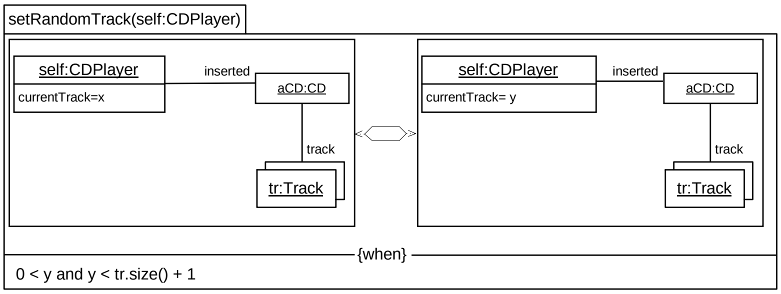


Fig. 4. Specification of `setRandomTrack` with QVT

Basically, for increasing the expressive power of graph transformations the same idea as in B can be applied. In QVT, variables can occur in RHS even if they do not occur in LHS. Consequently, the value of these fresh variables is not fixed anymore by the first step of the rule application and can be chosen non-deterministically. In order to get at least partial control over the values of these variables, QVT has added when-clauses to transformation rules. A when-clause contains constraints written in OCL. The constraint restricts the possible values not only for fresh variables used in RHS but for all elements in LHS and RHS.

The specification of `setRandomTrack()` shown in Fig. 4 takes advantage of the fresh variable `y` in RHS. The value of `y` is restricted in the when-clause what exactly captures the intended semantics.

4 Giving Graph Transformations a Loose Semantics

Although the integration of the when-clause is a necessary step to make graph transformations widely applicable and to overcome the determinism problem, this step is not sufficient. Another immanent problem of constructive languages remained unsolved. It is sometimes necessary to express in the contract that the implementations of the operation are allowed to change parts of the system state in an arbitrary way. If one puts this request to its very end, it means that in some cases the loose semantics of restrictive languages is needed.

In this section, we propose an extension of QVT that makes it possible to simulate the loose semantics of purely restrictive contracts written in OCL. These enrichments require a slight extension of QVT's notation to describe LHS and RHS.

4.1 Possible Side Effects of Restrictive Specifications

As argued in Sect. 2, the contract for `setNextTrack()` written in OCL does not exclude unintended side effects. These side effects can be classified as follows:

1. On object `self`, the values of the attributes not mentioned in the post-condition might have been changed.

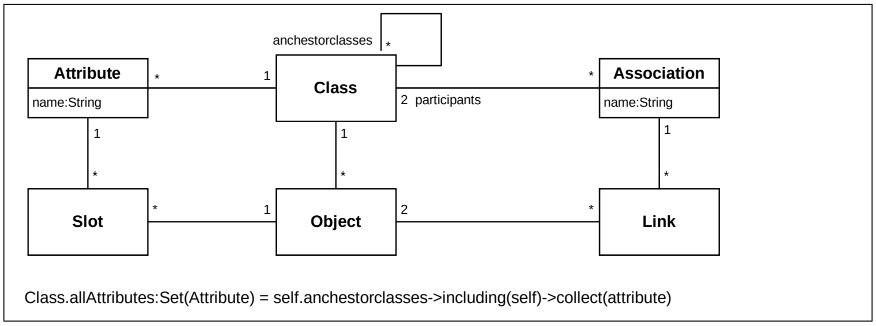


Fig. 5. Simplifed metamodel for states

2. The values of attributes of `CDPlayer`-objects different from `self` might have been changed.
3. The values of attributes of objects of other classes might have been changed.
4. An unrestricted number of objects of some classes might have been newly created.
5. An arbitrary number of existing objects except `self` might have been deleted.
6. An arbitrary number of links might have been created/deleted.

We will demonstrate in Sect. 4.3 how the contract for `setNextTrack()` shown in Fig. 3 had to be changed in order to capture each of these possible side effects. Beforehand, in the next subsection, the new constructs proposed for QVT, which are needed to simulate loose semantics, are summarized.

4.2 A Proposal for Extending QVT

Optional Creation/Deletion of Objects and Links. Graph transformation rules must be able to express that an object is optionally created or deleted. The same holds for links. So far, one can only specify that an object/link must have been created (deleted) by displaying the object/link in RHS but not LHS (in LHS but not in RHS). We propose to adorn an object/link in RHS with a question mark ('?') to mark its optional creation/deletion.

Note that it is a proven technique to adorn elements in LHS and RHS in order to modify the standard semantics of the rule. QVT and other graph transformation formalisms allow already to adorn elements with 'X' in order to express a negative application condition (NAC).

Placeholders to Denote Arbitrary Attributes/Classes. A more significant extension of graph transformations is the introduction of placeholders. Currently, QVT allows to describe the change of an attribute value only if the name of the attribute is known. One can, for example, not specify the reset of all attributes of type Integer to 0 unless all these attributes explicitly occur in the graph transformation rule.

We propose to use placeholders for attributes as a representation of arbitrary attributes. These placeholders appear in the same compartment of the object as normal attributes. In order to distinguish between normal attributes and placeholders, we start the name of the latter always with a backslash (\). This convention relies on the assumption that the name of normal attributes never starts with backslash. For example, if `\att` appears in the attribute compartment of an object, then it represents all attributes of this object (including attributes inherited from super-classes).

Sometimes, a placeholder should not represent all possible attributes but only some of them. To achieve this, we propose to use QVT's when-clause to define using OCL constraints which attributes are represented by which placeholders. Such OCL constraints, however, refer to the metamodel of UML object diagrams. To ease the understanding, we rely here on a simplified version of the official metamodel as shown in Fig. 5.

Furthermore, in order to distinguish easily OCL constraints referring to the metamodel from ordinary ones, we decided – slightly abusing OCL's official concrete syntax – to precede within OCL expressions each navigation on the metalevel with a backslash.

Besides placeholders for attributes there are also analogously defined placeholders for classes.

4.3 Realization of Possible Side Effects

We give examples on how possible side effects of OCL constraints presented in Sect. 4.1 can be simulated using our extension of QVT. In all cases, we start from the constructive specification of `setNextTrack()` shown in Fig. 3.

Other Attributes for self can change. A naive solution could be to explicitly list all attributes of object `self` in both LHS and RHS and to assign in RHS a fresh variable to the attribute.

This solution is first of all tedious to write down and in addition has the limits that were already discussed: In time of writing the contract, not all subclasses of `CDPlayer` might be known. Be aware that the QVT rule formulated in Fig. 3 is applicable even when `self` matches with an object whose actual type is not `CDPlayer` but a subclass of it. The core of the problem is, that, when writing the contract, we cannot predict which attributes the object `self` actually has.

The rule shown in Fig. 6 overcomes this principal problem. Each attribute of `self` is represented by placeholder `\attDiffCurrentTrack` as long as its name is different from 'currentTrack'. This is precisely described in the when-clause by an OCL constraint: For the actual class of `self` (which might be a subclass of `CDPlayer`) all valid declarations of attributes are collected. Note that attributes can have also been declared in one of the super-classes. The OCL constraint in the when-clause stipulates that the placeholder `\attDiffCurrentTrack` stands for any attribute as long as it is not named 'currentTrack' since this attribute cannot be changed in an arbitrary way. The value of `\attDiffCurrentTrack` in LHS is represented by variable `v`, which does not occur in the RHS. The new value `v'` in RHS shows that the value of the attribute matching with `\attDiffCurrentTrack` might have been changed during the execution of the operation.

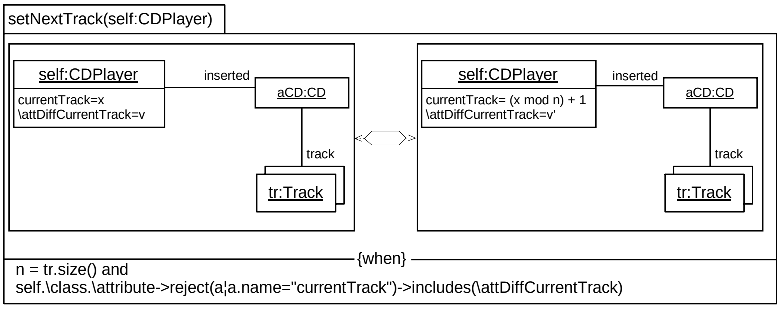


Fig. 6. Different attribute values for `self`

State of other CDPlayer-objects might change. This side effect is similar to the effect of changing the state of `self` and can be captured by applying the same technique to enrich the QVT transformation. A new object `other` is added to both LHS and RHS. In RHS, the value of the placeholder `\att` is changed to a possibly new value `v'`.

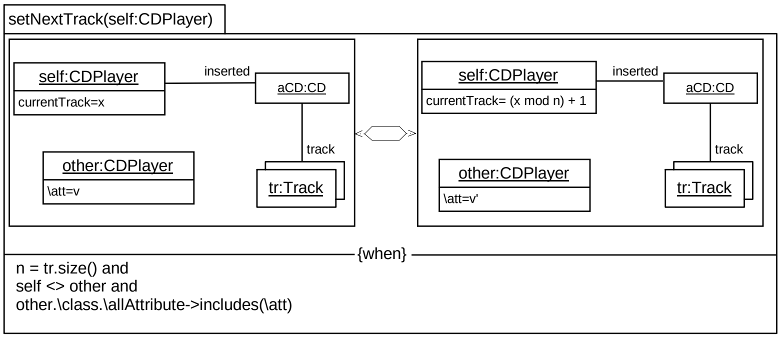


Fig. 7. Different attribute values for other objects of class `CDPlayer`

State of objects of other classes might change. In order to simulate state changes on objects of arbitrary classes different from `CDPlayer` (and its subclasses) placeholders for classes are needed. We have introduced the placeholder `\OtherClass` whose value is restricted by an appropriate constraint in the when-clause. The technique to change the state of objects of class `\OtherClass` is the same as the one exploited above to simulate the state change of `CDPlayer`-objects.

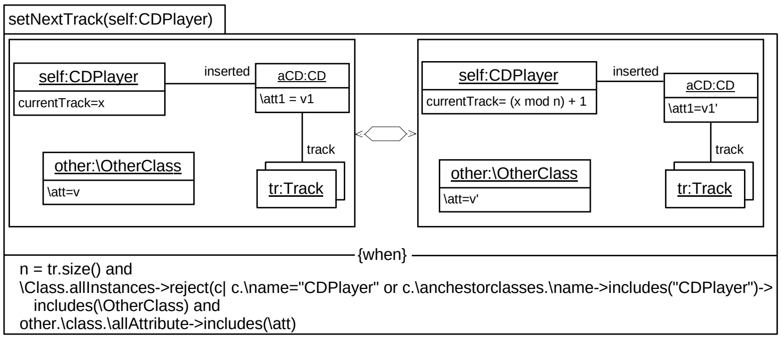

setNextTrack(self:CDPlayer)
self:CDPlayer
currentTrack=x
inserted
aCD:CD
\att1 = v1
track
tr:Track
other:\OtherClass
\att=v
self:CDPlayer
currentTrack= (x mod n) + 1
inserted
aCD:CD
\att1=v1'
track
tr:Track
other:\OtherClass
\att=v'
{when}
n = tr.size() and
\Class.allInstances->reject(c| c.\name="CDPlayer" or c.\anchestorclasses.\name->includes("CDPlayer")->
includes(\OtherClass) and
other.\class.\allAttribute->includes(\att)

Fig. 8. Different attributes for object of other classes

Objects different from self might have been deleted. It is not enough to add the question mark to the new object `other` (that represents an arbitrary object different from `self`). Unfortunately, the question mark must also be attached on all objects different from `self` that are explicitly mentioned in RHS (without such a question mark, the QVT semantics stipulates that all objects occurring in RHS are not deleted). In addition, also the multiobject `tr` might change since some of its elements are possibly deleted. Consequently, a new multiobject `tr1` is introduced in RHS, which – according to the when-clause – must be a subset of the original multiobject `tr`.

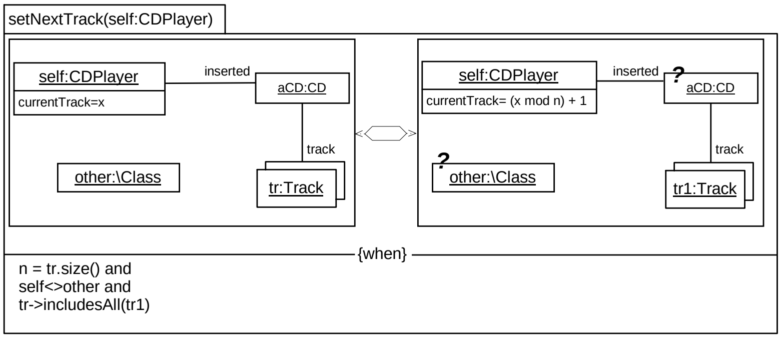


Fig. 9. Deletion of objects

Objects might have been created. Optional creation of arbitrarily many objects is expressed by adding a multiobject `other` to RHS. For each class, `other` represents the set of newly created objects. Furthermore, the multiobject `tr` might have been enlarged and became `tr1`.

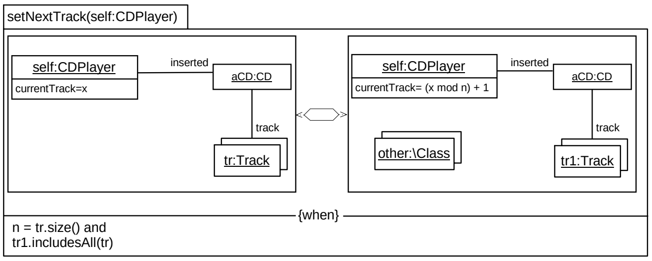


Fig. 10. Creation of objects

Links might have been created. For the optional creation of links, two arbitrary objects `o1`, `o2` are searched in LHS. The classes of `o1`, `o2` must be connected by an association with name `assoname`. RHS stipulates the optional creation of a corresponding link between both objects.

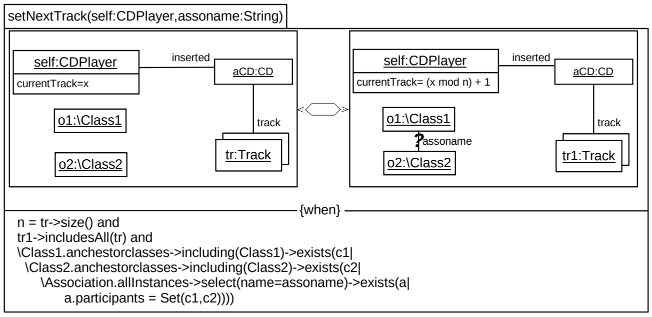

setNextTrack(self:CDPlayer,assoname:String)
self:CDPlayer
currentTrack=x
inserted
aCD:CD
o1:\Class1
track
o2:\Class2
tr:Track
self:CDPlayer
currentTrack= (x mod n) + 1
inserted
aCD:CD
o1:\Class1
?assoname
o2:\Class2
track
tr1:Track
{when}
n = tr->size() and
tr1->includesAll(tr) and
\Class1.anchestorclasses->including(Class1)->exists(c1|
\Class2.anchestorclasses->including(Class2)->exists(c2|
\Association.allInstances->select(name=assoname)->exists(a|
a.participants = Set(c1,c2))))

Fig. 11. Creation of links

Links might have been deleted. Analogously to the optional deletion of objects we mark also links that are deleted optionally with a question mark. Note, that the deletion of links might have be an effect on the multiobject `tr` the same way the deletion of objects has.

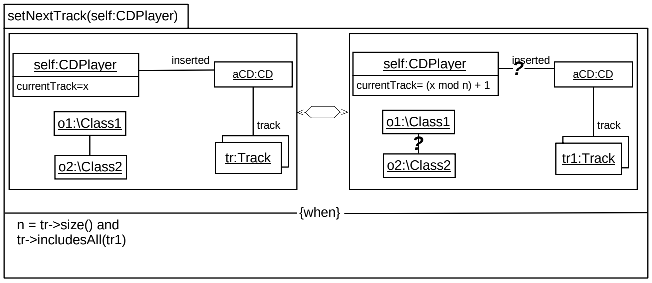


Fig. 12. Deletion of links

5 Conclusion and Future Work

In this paper, pros and cons of the two main behavior specification paradigms – constructive and restrictive style – are discussed. If restrictive languages do not provide provision for tackling the frame problem (such as OCL), then the specified contracts are comparably weak and do most often not capture the behavior intended by the user. Constructive languages suffer from the opposite problem as they sometimes prescribe too detailed the behavior and do not allow the freedom for variations among possible implementations. These two fundamental problems make it also very difficult to define a semantically preserving transformation from specifications of restrictive specification languages into specifications written in a constructive language, or vice versa.

Graph transformations can be used as a basically constructive specification language but it is sometimes also possible to pursue a restrictive specification style. Contracts given in form of a graph transformation rule have the advantage of being easily accessible by humans due to the visual format. In many cases, constructive contracts are intended and constructive contracts work well. For the case that a purely constructive semantics is not appropriate, we have given in Sect. 4 a catalog of proposals to enrich a graph transition rule so that the intended behavior is met. This approach to adapt the semantics of the rule more to the loose semantics of restrictive languages is very flexible since the user has the possibility to traverse the metamodel with OCL constraints.

A lot of work remains to be done. First of all, the proposed formalism of extended graph transformations should be implemented by a tool to resolve all the small problems that can only be recognized if a tool has to be built. In order to become confident in the formal semantics of the formalism, an evaluator needs to be implemented that can decide for any contract and any given state transition whether or not the transition conforms to the contract.

Once such a tool is available, it should be applied on bigger case studies showing or disproving the appropriateness of the proposed formalism for practical software development.

References

1. Bertrand Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, October 1992.
2. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, second edition, 1997.
3. Levi Lucio, Luis Pedro, and Didier Buchs. A methodology and a framework for model-based testing. In Nicolas Guelfi, editor, *Rapid Integration of Software Engineering Techniques, First International Workshop, RISE 2004, Luxembourg-Kirchberg, Luxembourg, November 26, 2004, Revised Selected Papers*, volume 3475 of *LNCS*, pages 57–70. Springer, 2004.
4. J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, pages 463–502, 1969.

5. Claudia Ermel and Roswitha Bardohl. Scenario animation for visual behavior models: A generic approach. *Software and Systems Modeling (SoSym)*, 3(2):164–177, 2004.
6. Lars Grunske, Leif Geiger, Albert Zündorf, Niels van Eetvelde, Pieter van Gorp, and Dániel Varró. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering. Springer, 2005.
7. OMG. Revised submission for MOF 2.0, Query/Views/Transformations, version 1.8. OMG Document ad/04-10-11, Dec 2004.
8. Reiko Heckel, Mercè LLabrés, Hartmut Ehrig, and Fernando Orejas. Concurrency and loose semantics of open graph transformation systems. *Mathematical Structures in Computer Science*, 12:349–376, 2002.
9. Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.
10. Andy Schürr. Adding graph transformation concepts to UML's constraint language OCL. *Electronic Notes in Theoretical Computer Science, Proceedings of UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, 44(4), 2001.
11. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
12. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1992.
13. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for java. Technical Report TR 98-06-rev28, Department of Computer Science, Iowa State University, 2005. Last revision July 2005, available from www.jmlspecs.org.
14. A. Borgida, J. Mylopolous, and R. Reiter. ...And Nothing Else Changes: The Frame Problem in Procedure Specifications. In *Proceedings of ICSE-15*, pages 303–314. IEEE Computer Society Press, 1993.
15. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In Lionel Briand and Clay Williams, editors, *Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
16. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, August 1996.
17. Thomas Baar. Non-deterministic constructs in OCL – what does any() mean. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *Proc. 12th SDL Forum, Grimstad, Norway, June 2005*, volume 3530 of *LNCS*, pages 32–46. Springer, 2005.
18. ClearSy. Atelierb homepage. http://www.atelierb.societe.com, 2005.
19. Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

Author Index