Model Transformation from VisualOCL to OCL using Graph Transformation

Karsten Ehrig [1] Jessica Winkelmann [2]

Technical University of Berlin, Germany

Abstract

In this paper we present a model transformation from a visual representation (VisualOCL) of the Object Constraint Language (OCL) to the textual one using graph transformation. Starting from VisualOCL diagrams, we show how their underlying abstract syntax can be modeled by typed attributed graphs and converted into an OCL string representation using graph transformation rules.

Key words: ocl, visual ocl, graph grammar, graph transformation, model transformation

1 Introduction

For defining models and metamodels, UML [OMG04] has been established as a well known standard for software system development, especially for designing and documenting software and systems.

For extending the expressive power, the *Object Constraint Language (OCL)* [OMG03] is defined as part of UML. OCL is a textual language to specify additional constraints about objects in a UML model.

VisualOCL was developed in [BKPPT01,KTW02] directly as a visualization of OCL and is meant as an alternative notation to the textual one. VisualOCL provides additional visual information to the user which increase the usability of a purely textual language (OCL) that is used together with the visual language UML. VisualOCL follows the UML notation as far as possible. This makes a direct integration of OCL into UML easier. Like OCL, VisualOCL is a formal, typed and object-oriented language.

The abstract syntax of VisualOCL is precisely defined by graph transformation [Win05] on one hand, and metamodeling on the other hand.

[1] Email: karstene@cs.tu-berlin.de
[2] Email: danye@cs.tu-berlin.de

For VisualOCL a visual editor was developed as an Eclipse plug-in [Vis04] which captures all main concepts of VisualOCL. The user can draw and edit a VisualOCL constraint using the features of Eclipse [Ecl] and the Graphical Editing Framework [GEF].

The aim of our paper is to define a model transformation from VisualOCL to OCL using graph transformation which could be directly executed in the VisualOCL editor plug-in. Moreover a tool chain is described to transform the VisualOCL diagram into an attributed graph via the Graph Exchange Language (GXL) [GXL] which is transformed using model transformation rules to a semantically equivalent OCL string.

Section 2 gives an overview on the VisualOCL language. In Section 3, we describe the model transformation from VisualOCL to OCL and in Section 4 we show how the model transformation could be implemented in the VisualOCL editor plug-in using the graph transformation engine AGG [AGG].

2 VisualOCL

In this section, a short introduction to VisualOCL is given using some simple examples.

VisualOCL is a visualization of the complete OCL 2.0 release 1.5 [OMG03]. VisualOCL follows the UML notation and its graphical representation as far as possible. VisualOCL is a formal, typed and object oriented language. New data types and operations such as collections and operations like *forall*, *select*, *union* etc. are represented by simple but meaningful graphics. Logical expressions are denoted as Peircian graphs using nested boxes to express disjunctions and conjunctions.
A VisualOCL diagram describes additional constraints about objects in an UML model. The class diagram in Fig. 1 provides the structural information for the examples.

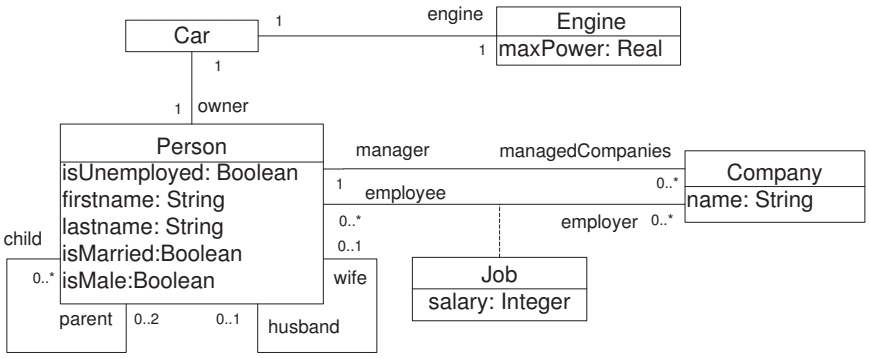


Fig. 1. Class Diagram Example

The following examples are presented in OCL and as VisualOCL diagrams. The diagram captions express the constraint in natural language.

An OCL constraint is visualized as a rounded rectangle with two sections, the context section and the body section. The context section contains the keyword *context* followed by the type name of the model element (mostly a class or method) of the constraint followed by the kind of the constraint e.g. *inv*, *pre*, *post* or *def*. Thus, the context is specified as in OCL. In the body section, the body of the constraint is visualized. If the body is a navigation expression, it may have a condition section which contains textual sub conditions of the constraint declared, using variables defined in the body. If there is a condition section, it is separated from the rest of the body by a dashed line.

The variable *self* is used like in OCL and is always an instance of the type of the context. If available, the navigation starts at this instance. The visualization of an object corresponds to that in UML communication diagrams. The attribute value of an object can be referred to by a variable. This is useful e.g. if it shall be compared with other attribute values.

In Fig. 2, the variable y refers to the first name of a person. The lastname is compared with the first name. Both have to be different for each person.

- `context Person inv: firstname <> lastname`

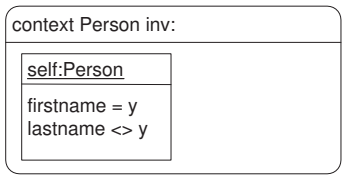


Fig. 2. Invariant 1: Each person has a first name different from the last name.

Object navigation is also visualized as in UML communication diagrams. For the navigation, the role name of the opposite association end is used. In the case of unambiguous navigation, the role name can be left out. This is always the case, if there exists only one association between the classes. The navigation result is the set of objects on the opposite end of a link and has the multiplicity as defined for the corresponding association in the class diagram. The navigation on any associations always starts at object *self* if it exists, or at the context object defined otherwise. If there is only one object of the context type, this can also be used as starting point. Navigations can end at objects, association classes, attribute values, and method or operation calls.

Above we described navigation expressions, now we continue with several other kinds of VisualOCL expressions. A *let expression* defines a variable which can be used in a constraint after its definition. It is depicted by two frames, a *let* frame and an *in* frame. The *let* frame contains the visualized definition of the variables. There is a separate frame for each variable where the name of the variable is depicted in the upper left corner and below the definition of the variable value follows. Inside the *in* frame a normal constraint is described which uses the variables defined above. A *let expression* is only

known in the constraint in which it was defined.
The *if-then-else* frame contains three sections, the *if* section describes the *if* condition, the *then* section describes the *then* part and the *else* section describes the *else* part. While the *if* section has to be a boolean expression, each of the other two sections can contain any VisualOCL expression. Two objects in different VisualOCL expressions are identical, if they have the same name.

- `context Person inv:`
  `let carSize : Real = car.engine.maxPower in`
  `if (isUnemployed = true )`
  `then carSize < 1.0 else carSize >= 1.0`

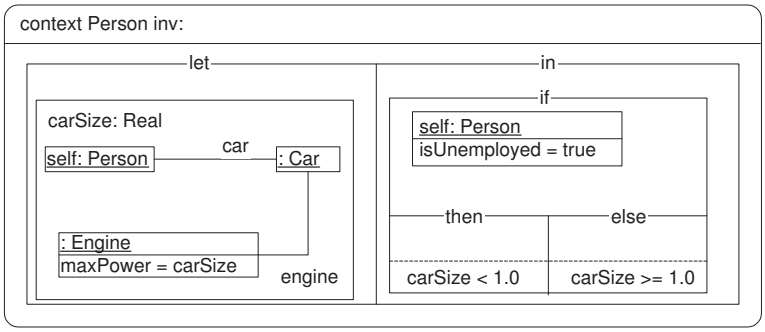
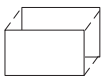


Fig. 3. Invariant 2: Unemployed people drive small cars.

Furthermore, there are Boolean expressions with operands like *implies*, *or*, *xor* etc. represented by nested boxes.

Collections, like sets, bags, and sequences, are predefined types in OCL. The collection type is an abstract type with three concrete subtypes: *Set*, *Bag* and *Sequence*. These types are visualized as follows:

Set:  Bag:  Sequence: 

Collections have a large number of predefined operations, simple operations e.g. *isEmpty()* and *notEmpty*, a large variety of set operations like *includes*, *diff*, etc., or iterator operations like *select*, *forall*, *exists* etc. Simple operations are directly annotated at collections. The visualization of set and iterator operations is more complex. For example, operation *exists* checks if a constraint is satisfied for at least one element in a collection. It has one iterator and an *exists* frame in which the property of the *exists* operation is visualized. The *exists* operation returns a Boolean value. On the right of the frame in Figure 4, the ∃-operator and the iterator are depicted. Operation *select* specifies a subset of a collection. In Fig. 4, a shortcut of a *select* operation is shown. If the iterated value is just an attribute of a collection element, the shortcut for iterator operations can be used. On the right of the collection representation, the operator name is depicted. An unlabeled arrow targets at the resulting

collection.

- `context Company inv: Company->exists(c |c.employee->select(p | p.isFemale and p.isMarried)->isEmpty())`

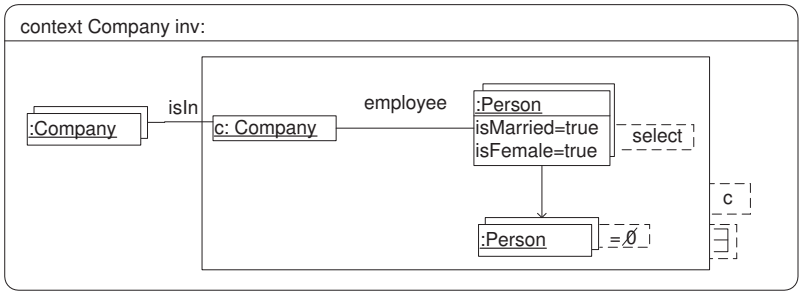


Fig. 4. Invariant 3: There is a company without female married employees.

An *implies* expression is visualized in an *implies* frame. Anything above or left of the keyword *implies* describes the premise. When this premise is true, it implies the conclusion denoted right or below of *implies*. Both sections may contain any boolean expression. Note that the conclusion part of the implies operation in Fig.5 contains a navigation expression where four subexpressions are combined by *and*. Please note that this is possible withoutboxing each expression.

- `context Person inv:`
  `(self.isMarried=true and self.isMale=true) implies`
  `(self.husband->isEmpty() and self.wife->notEmpty() and`
  `self.wife.isMarried = true)`

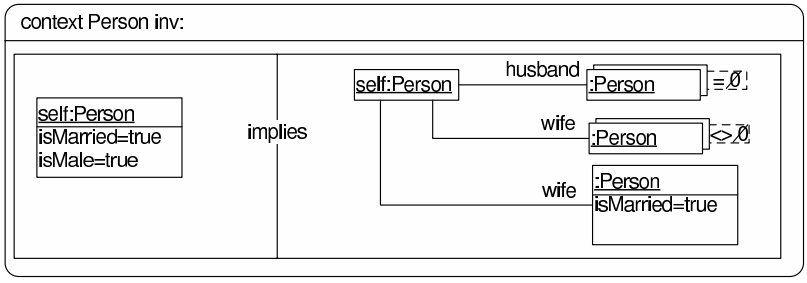


Fig. 5. Invariant 4: Married men have a married wife and no husband.

3 From VisualOCL to OCL

This section describes the transformation from an instance graph of the VisualOCL type graph to an equivalent string containing a textual OCL representation of that instance graph. Fig.6 represents a model transformation by graph transformation. The source and target models are graphs. Performing

model transformation by graph transformation means to take the underlying structure of a model as graph, and to transform it according to certain transformation rules. The result is a graph which shows the underlying structure of the target model. The abstract syntax graphs of the source models are instance graphs over a type graph T_s, the abstract syntax graphs of the target models are instance graphs over a type graph T_t. T_s and T_t are subgraphs of type graph T. Starting the model transformation with instance graph G_s typed over T_s it is also typed over T. The intermediate graphs are typed over T which may contain types and relations which are neither in T_s nor in T_t but needed for the transformation process.

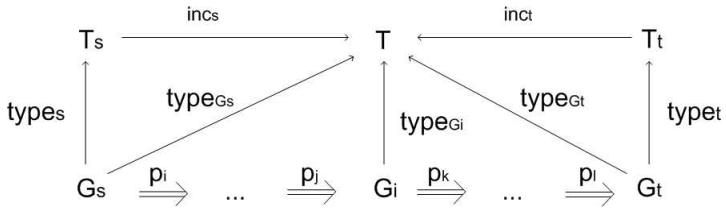


Fig. 6. Model Transformation Process

The type graph of the source models is the VisualOCL type graph defined in [Win05], the type graph of the target model consists of only one type node, the OCLString. The type graph of the intermediate models contains the source type graph and the one type node of the target type graph. It also contains relations between many node types from the VisualOCL type graph to the node type OCLString.
Now we describe the rules that model the transformation. The type graph of VisualOCL and the transformation rules were defined with the AGG graph grammar editor.

In the following a subset of these rules is explained.

A VisualOCL constraint is represented as a node of type Constraint with a name, a kind, an edge to the constrained element and an edge to the body element, that is a VOCLExp (NavExp, IfExp, BooleanExp or LetExp). The rules are divided into several layers. First the body of the VisualOCL diagram is transformed and then the complete constraint is build (combining the body expression with the constraint name, kind etc.). The body can be a NavExp, IfExp, BooleanExp or LetExp. IfExp, BooleanExp and LetExp can contain other VOCLExps, so at first the NavExps have to be transformed and then, in a higher layer, the other VOCLExps.

NavExps consists of roles that can be ClassifierRoles, representing single instances, and SetRoles, BagRoles or SequenceRoles, representing collections of instances. The roles can be connected by association roles or collection operations, like *union* operation, that return other roles. Each NavExp has a start and an end role. For each role along the navigation expression a new OCLString is build from the existing string of the last role and the association end name of the associarion to the current role.

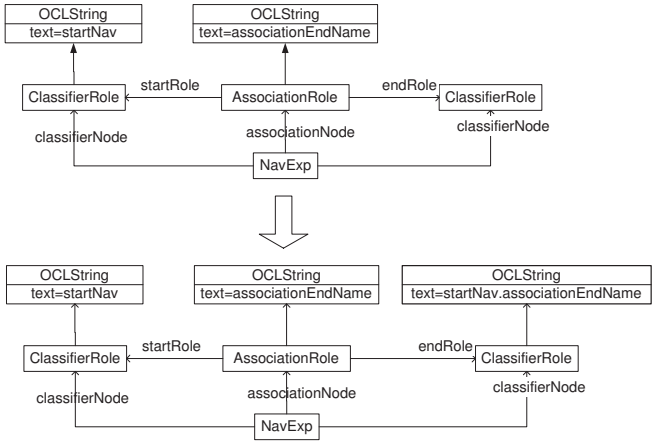


Fig. 7. Creation of the OCLString in a navigation expression

In Fig. 8 the rule for creating the OCLString of the result role of a complex collection operation is shown. The string is built of the OCLString representing the start navigation to the role on which the operation is assigned, the operation name, and the corresponding OCLString of the argument role. If the NavExp contains a loop operation, like *select*, that returns a role, the creation of the OCLString is similar to that described above, see Fig.9.

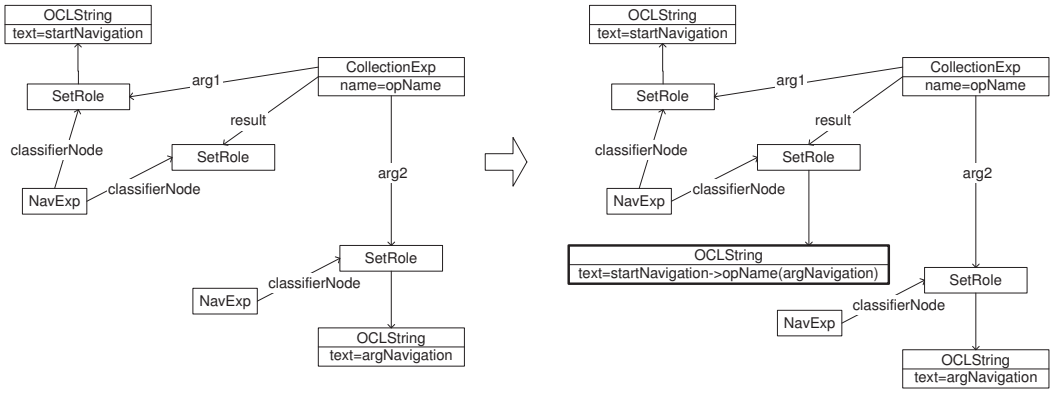


Fig. 8. Creation of the OCLString for a complex collection operation

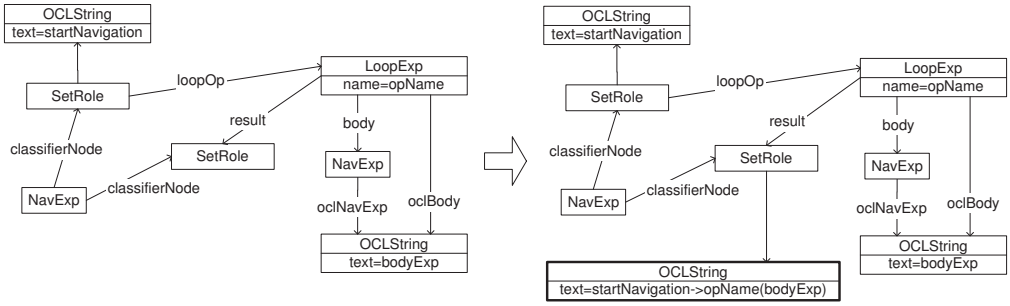


Fig. 9. Creation of the OCLString for a loop operation

After applying the rules described before, each role of all NavExps has

edges to corresponding OCLString(s). The navigation can end at a role (in case of equivalent navigation paths like *self.ass1.ass2=self.ass3.ass4*), at an attribute or method result value (navigation expressions like *self.ass1.attr1=true* or *self.method1()=false*), or at a simple collection operation (navigation expressions like *self.ass1.ass2->isEmpty()*). The rules for the creation of the corresponding OCLStrings are shown in Fig.10 and Fig.11. If the navigation ends at a collection operation or loop expression with boolean result, the OCLString is built in a similar way.

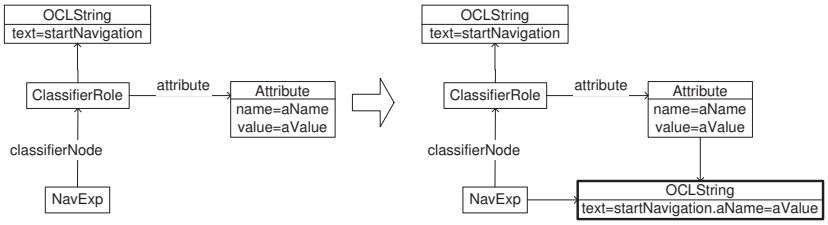


Fig. 10. Creation of the OCLString for an attribute as navigation end

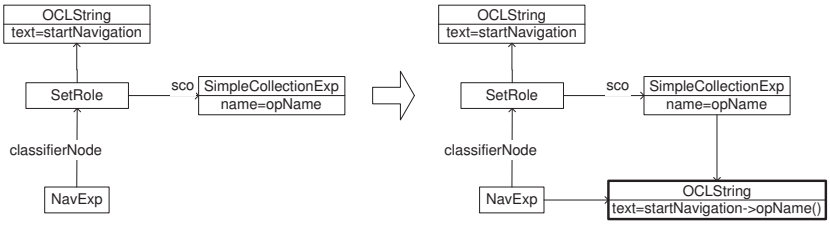


Fig. 11. Creation of the OCLString for a simple collection operation as navigation end

Now the NavExp can have edges to several OCLStrings, describing an attribute navigation, method navigation or a navigation to a simple collection expression. These strings have to be combined by "and", see the rule in Fig. 12.

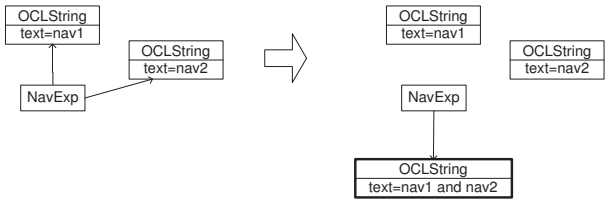


Fig. 12. Creation of the OCLString for a NavExp

Now the navigation expressions are translated into OCLStrings (containing an equivalent textual representation) and the other VisualOCL expressions can be transformed. Therefore the parts (like *if* or *let* part) or arguments (two arguments of a logical operation) of the expression have to be transformed (they must have an edge to a corresponding OCLString). Then the new OCLString can be built by combining the strings with the particular operator name. The

rule for a boolean operation, like *or*, *implies* etc. is shown in Fig.13. The rules creating the OCLString for an IfExp or LetExp are analog.

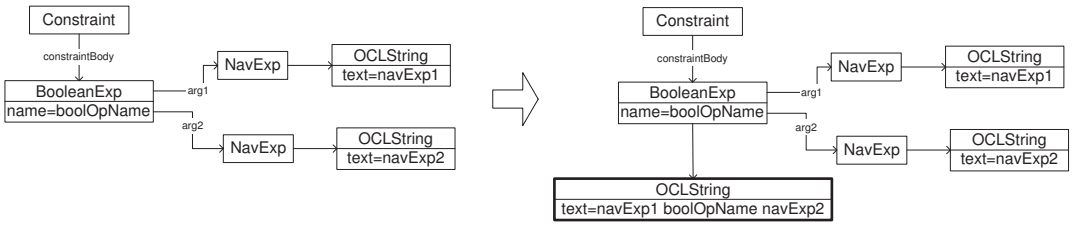


Fig. 13. Creation of the OCLString for a BooleanExp

The last step is to create the OCLString for the constraint which contains the context definition and the context body which was already created, see Fig.14.

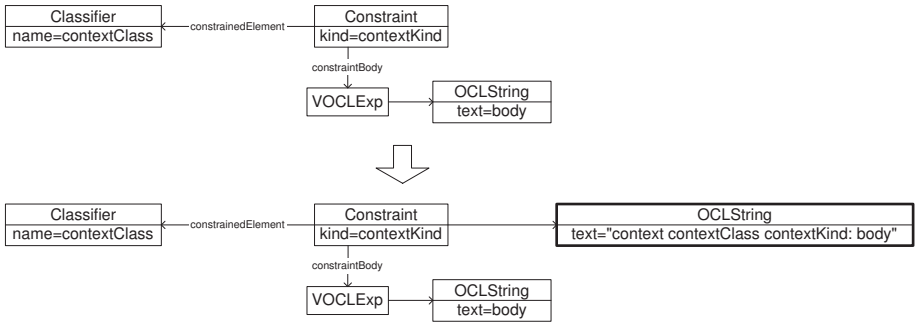


Fig. 14. Creation of the complete OCLString of the constraint

If the constraint node has no edge to a node of type VOCLExp with an equivalent OCLString node an OCLString node with an error message is created for the constraint node.

Now rules that delete the edges to the OCLString nodes and all OCLString nodes which are not connected to a node of type Constraint can be applied. Thereafter each node and edge except the OCLString can be deleted.
The resulting instance contains one node of type OCLString whose text attribute holds a string that is equivalent to the constraint expressed by the VisualOCL type graph instance which was the source graph.

3.1 Reverse Transformation

The reverse transformation from an OCLString to VisualOCL is possible in principal, but not unique, because there could exist different representations for an OCLString in VisualOCL. For instance, consider the constraint
context Person inv: self.isMarried=true implies self.age>=18
that could be translated to the following visualizations: In the VisualOCL constraint on the left hand side, the attribute condition *age>=18* is included in the classifier role whereas this expression is translated to a variable assignment *age=x* and the condition *x>=18* on the right hand side in Fig. 15. A unique

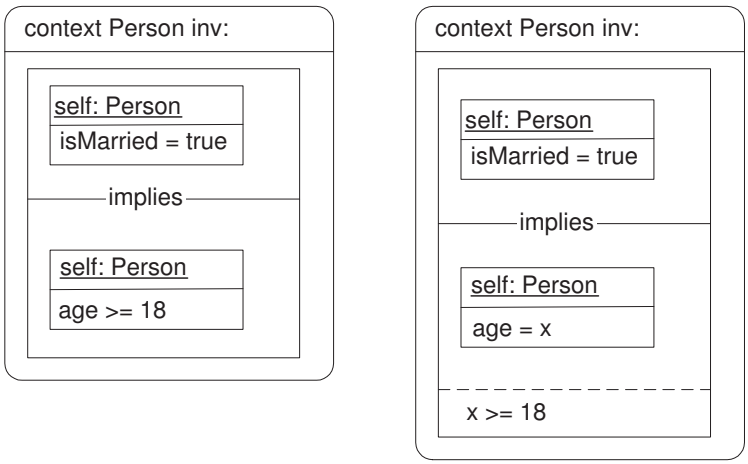


Fig. 15. reverse transformation

reverse transformation for this example could be provided by default setting which are not yet included in the VisualOCL specification [Win05].

Although the automatic inversion of the model transformation rules is not possible, a new model transformation graph grammar for the inverse transformation could be created. An alternative model transformation could be the transformation to an OCL meta model instance.

4 Implementation

The model transformation from VisualOCL to OCL is implemented in a visual editor plug-in for Eclipse [Vis04] which captures all main concepts of VisualOCL. The user can draw and edit a VisualOCL constraint using the features of Eclipse [Ecl] and the Graphical Editing Framework [GEF].

Fig. 16 shows an overview of the model transformation in the VisualOCL editor plug-in:

- A VisualOCL diagram presented in the editor, is saved in a file with the extension *.vocl*.
- A *.vocl* file is transformed into a GXL file by an XSL transformation provided by file *vocl2gxl.xsl*.
- The GXL file is imported as host graph in the existing model transformation system in AGG.
- The model transformation rules are applied to the VisualOCL graph in AGG. The resulting AGG graph contains an *OCLString* with a text attribute (*text*) holding the converted OCL string.
- The converted OCL string is reimported in the VisualOCL editor plug-in.

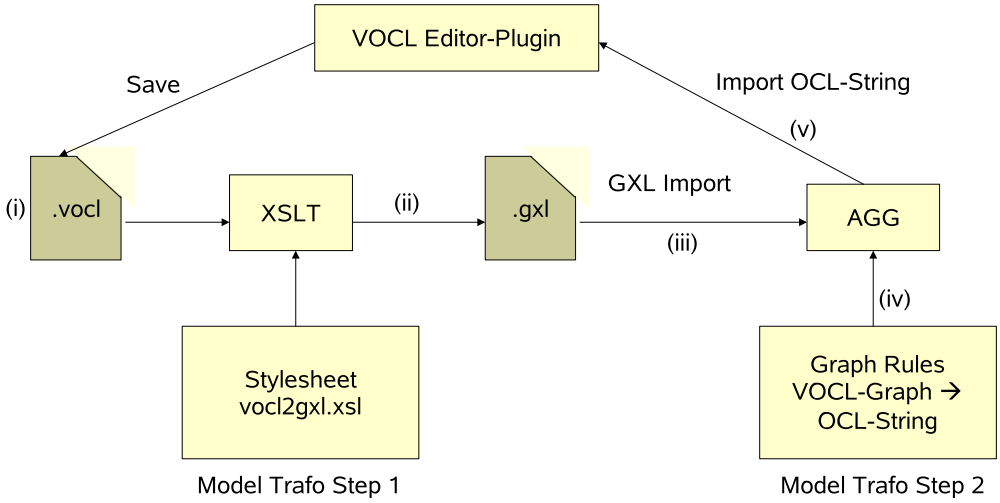


Fig. 16. Implementation of the Model Transformation

Step 1: XSL Transformation

The underlying model of the VisualOCL editor plug-in was generated with the Eclipse Modeling Framework (EMF) [EMF] based on an EMF meta model class diagram which corresponds to the type graph of VisualOCL [Win05]. The editor diagram is stored in a XML [XML] based format (*.vocl* file) typed over the EMF meta model. With stylesheet format transformation [XSL] rules given in the file *vocl2gxl.xsl* the *.vocl* file is converted into the graph exchange format (GXL) [GXL]. To allow a simple format transformation the EMF meta model classes and associations have an one to one correspondence to the VisualOCL type graph nodes and edges.

Step 2: Graph Transformation

The converted GXL file is imported as instance graph into the existing model transformation system in the graph transformation engine AGG [AGG]. The transformation process leads to a *OCLString* graph node with the converted OCL string expression which is reimported in the VisualOCL editor plug-in.

VisualOCL editor plug-in

Fig. 17 shows the example constraint from Fig. 3 in the VisualOCL editor plug-in which is transformed into the OCL string shown in Fig. 18.

Java Code Converter

In addition the conversion from VisualOCL to OCL was realized with plain Java code. Converting of simple navigation expression was straightforward using plain Java code but including the hierarchy of *if*, *let* and *boolean* expression

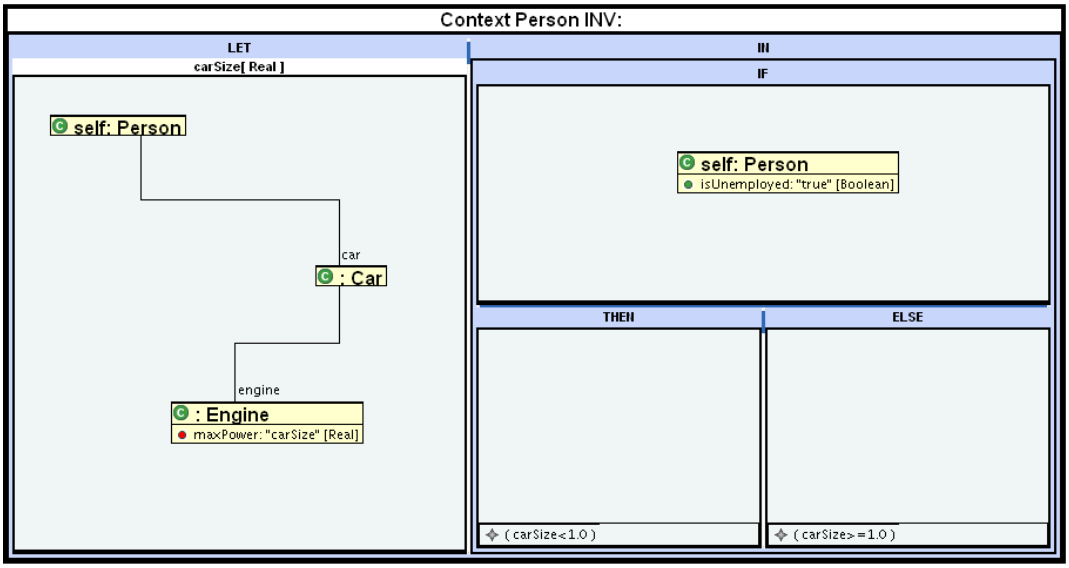


Fig. 17. Example constraint in the VisualOCL editor plug-in

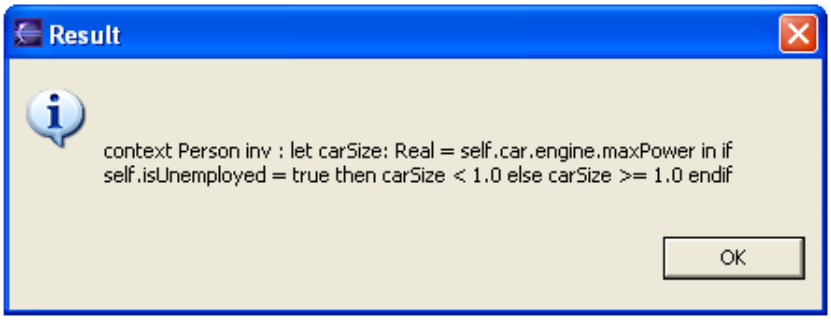


Fig. 18. Converted OCL string shown in the VisualOCL editor plug-in

results in complex Java code of more than 1200 lines of code. Especially the implementation of a search algorithm in navigation expressions was a none trivial task.

For this reason the implementation of the model transformation from VisualOCL to OCL using graph transformation has a big advantange using the visual definition of model transformation rules with the AGG GUI. Furthermore AGG provides consistency checks like termination and critical pair analysis [EEdL+05,EEPT05]. Due to the complexity of our model transformation consistency checks of AGG are not applicable to the model transformation up to now.

Comparing the computation time, the Java code converter took less then one second compared with about three seconds for the model transformation via graph transformation rules (computed on a notebook with 600MHz processor).

5 Conclusion

The model transformation from VisualOCL to OCL closes the gap between the visual and textual representations of OCL. VisualOCL provides an intuitive and easy way for visual modeling in OCL. With the model transformation shown in this paper a visual OCL constraint could be used in various applications working with the textual representation of OCL, such as the extension of Eclipse EMF by OCL, provided by the *Object Constraint Language Library* from the University of Kent [AP03]. This working group has also introduced Constraint Diagrams in [Ken97] as a visual notation for expressing logical constraints in an object-oriented system. They may be used as a modeling notation in their own right [HS05] and are independent of the UML and OCL. However, they can be used in the context of the UML as an alternative to (or a visualization of) the OCL. The basic diagrams have been fully formalized [FFH05], but a complete modeling framework involving the notation is still under development. It may be interesting to compare these visualizations and the transformations to the corresponding OCL strings.

Using graph transformation for model transformation has many advantages then using a Java coded converter resulting in a high level description technique for visual definition of model transformation rules. Furthermore, consistency checks like termination and critical pair analysis [EEdL+05,EEPT05] are available for graph transformation techniques although they are not applicable to all graph transformation systems yet.

References

[AGG] AGG – version 1.2.6. http://tfs.cs.tu-berlin.de/agg.

[AP03] D. Akehurst and O. Patrascoiu. Object constraint language library. http://www.cs.kent.ac.uk/projects/ocl, 2003.

[BKPPT01] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualization of OCL using Collaborations . In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language*, LNCS 2185, pages 257 – 271. Springer, 2001.

[Ecl] Eclipse – version 3.0. http://www.eclipse.org.

[EEdL+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 214–228. Springer Verlag, 2005.

[EEPT05] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Foundamentals of Algebraic Graph Transformation.* EATCS Monographs in TCS, Springer to appear, 2005.

[EMF] Eclipse Modeling Framework (EMF) – version 2.0. http://www.eclipse.org/emf.

[FFH05] A. Fish, J. Flower, and J. Howse. The semantics of augmented constraint diagrams. In *Journal of Visual Languages and Computing*. Elsevier, to appear, 2005.

[GEF] Eclipse Graphical Editing Framework (GEF) – version 3.0. http://www.eclipse.org/gef.

[GXL] Graph exchange language – version 1.0. http://www.gupro.org/gxl.

[HS05] J. Howse and S. Schuman. Precise visual modelling. In *Journal of Software and System Modeling*. Springer, to appear, 2005.

[Ken97] S. Kent. Constraint diagrams: Visualizing invariants in object oriented modelling. In *In Proceedings of OOPSLA97*, pages 327 – 341. ACM Press, 1997.

[KTW02] C. Kiesner, G. Taentzer, and J. Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. Technical Report 2002/23, Technical University of Berlin, 2002.

[OMG03] OMG. OCL 2.0 specification, revision 1.6. http://www.omg.org, 2003.

[OMG04] OMG. UML 2.0 specification. http://www.omg.org, 2004.

[Vis04] VisualOCL: Editor plugin for Eclipse. http://tfs.cs.tu-berlin.de/vocl/, 2004.

[Win05] Jessica Winkelmann. Specification of VisualOCL: A Visualisation of the Object Constraint Language. Master's thesis, TU Berlin, 2005. (in German).

[XML] Extensible markup language (xml). http://www.w3.org/xml.

[XSL] Extensible stylesheet language transformation (xslt). http://www.w3.org/TR/xslt.