# Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars

Jessica Winkelmann<sup>1</sup> Gabriele Taentzer<sup>2</sup>

Department of Computer Science Technical University of Berlin Germany

Karsten Ehrig<sup>3</sup>

Department of Computer Science, University of Leicester, UK

Jochen M. Küster<sup>4</sup>

IBM Zurich Research Laboratory CH-8803 Rüschlikon, Switzerland

#### Abstract

The meta modeling approach to syntax definition of visual modeling techniques has gained wide acceptance, especially by using it for the definition of UML. Based on class structures and well-formedness rules, which could be formalized by OCL, the abstract syntax of visual modeling techniques is defined in a declarative way. Since meta-modeling is non-constructive, it does not provide a systematic way available to generate all possible meta model instances. But for example, when developing model transformations, it is desirable to have a large set of valid instances at hand to perform large-scale testing. In our approach, an instance-generating graph grammar is automatically created from a given meta model. This graph grammar ensures correct typing and cardinality constraints. To satisfy also the given OCL constraints, well-formedness checks have to be done in addition. As a first step, a restricted form of OCL constraints can be translated to graph constraints which are to be checked during the instance generation process.

Key words: OCL, meta model, graph grammar, UML

This paper is electronically published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs

<sup>&</sup>lt;sup>1</sup> Email: danye@cs.tu-berlin.de

<sup>&</sup>lt;sup>2</sup> Email: gabi@cs.tu-berlin.de

<sup>&</sup>lt;sup>3</sup> Email: karsten@mcs.le.ac.uk

<sup>&</sup>lt;sup>4</sup> Email: jku@zurich.ibm.com

## 1 Introduction

Meta modeling is a wide-spread technique to define visual languages, with the UML [UML03] being the most prominent one. Despite several advantages of meta modeling such as ease of use, the meta modeling approach has one disadvantage: It is not constructive i. e. it does not offer a direct means of generating instances of the language. This disadvantage poses a severe limitation for certain applications. For example, when developing model transformations, it is desirable to have a large set of valid instance models available for large-scale testing. Producing such a large set by hand is tedious. In the related problem of compiler testing [BS97] a string grammar together with a simple generation algorithm is typically used to produce words of the language automatically. Generating instance-generating graph grammars for creating instances of meta models automatically can overcome the main deficit of the meta modeling approach for defining languages. The graph grammar introduced in [EKTW05] ensures cardinality constraints, but OCL constraints for the meta model are not considered until now. In this paper we present the main concepts of automatic instance generation based on graph grammars by an example. In addition, we show how restricted OCL constraints can be translated to equivalent graph constraints. The restricted OCL constraints that can be translated can express local constraints like the existence or non-existence of certain structures (like nodes and edges or subgraphs) in an instance graph. Positive ones have to be checked after the generation of a meta model instance, negative graph constraints can be checked during the generation. They can be transformed into application conditions for the corresponding rules, as defined in [EEHP04].

We first introduce meta models with OCL constraints in Section 2. Section 3 presents the main concepts for automatic generation of instances from meta models using the graph grammar approach. The generation process is illustrated at a simplified statechart meta model. We use graph transformation with node type inheritance [BEdLT04] as underlying approach. In Section 4 we explain how restricted OCL constraints can be translated into graph constraints. We conclude by a discussion of related and future work.

# 2 Meta Models with OCL-Constraints

Visual languages such as the UML [UML03] are commonly defined using a meta modeling approach. In this approach, a visual language is defined using a meta model to describe the abstract syntax of the language. A meta model can be considered as a class diagram on the meta level, i. e. it contains meta classes, meta associations and cardinality constraints. Further features include special kinds of associations such as aggregation, composition and inheritance as well as abstract meta classes which cannot be instantiated.

Each instance of a meta model must conform to the cardinality con-

straints. In addition, instances of meta models may be further restricted by the use of additional constraints specified in the Object Constraint Language (OCL) [Obj05].

Figure 1 shows a slightly simplified statechart meta model (based on [UML03]) which will be used as running example. A state machine has one top CompositeState. A CompositeState contains a set of StateVertices where such a StateVertex can be either an InitialState or a State. Note that StateVertex and State are modeled as abstract classes. A State can be a SimpleState, a CompositeState or a FinalState. A Transition connects a source and a target state. Furthermore, an Event and an Action may be associated to a transition. Aggregations and compositions have been simplified to an association in our approach but they could be treated separately as well. For clarity, we hide association names, but show only role names in Figure 1. The association names between classes StateVertex and Transition are called source and target as corresponding role names. The names of all other associations are equal to their corresponding role names. Since we want to concentrate on the main concepts of meta models here, we do not consider attributes in our example.

The set of instances of the meta model can be restricted by additional OCL constraints. For the simplified statecharts example at least the following OCL constraints are needed:

- (i) A final state cannot have any outgoing transitions: context FinalState inv: self.outgoing->size()=0
- (ii) A final state has at least one incoming transition: context FinalState inv: self.incoming->size()>=1
- (iii) An initial state cannot have any incoming transitions: context InitialState inv: self.incoming->size()=0
- (iv) Transitions outgoing InitialStates must always target a State: context Transition inv: self.source.ocllsTypeOf(InitialState) implies self.target.ocllsKindOf(State)



Fig. 1. Meta Model for Statecharts

# **3** Generating Statechart Instances

In this section, we introduce the idea of an instance-generating graph grammar that allows one to derive instances of a meta model in a systematic way. Given an arbitrary meta model, the corresponding instance-generating graph grammar can be derived by creating specific graph grammar rules, each one depending on the occurrence of a certain meta model pattern. The idea is to associate to a specific meta model pattern a graph grammar rule that



Fig. 2. Example Grammar Rules 1

creates an instance of the meta model pattern under certain conditions. An instance-generating graph grammar also requires a start graph and a type graph. The start graph will be the empty graph and the type graph is obtained by converting the meta model class diagram to a type graph. Given a concrete meta model, assembling the rules derived, the type graph created and the empty start graph will lead to an instance-generating graph grammar for this meta model. For a detailed description see [EKTW05]. Overall, we use the concept of layered graph grammars [EEdL<sup>+</sup>05] to order rule applications. In the following, we describe the rules that we derive for the meta model of state machines (see Figure 1).

First, we will get a create rule for each non-abstract class within the meta model, allowing us to create an arbitrary number of instances of all non-abstract classes. The rules of layer 1 are applied *arbitrarily often*, meaning that layer 1 does not terminate and has to be interrupted by user interaction or after a random time period. For the sample meta model we get the rules createStateMachine, createCompositeState, createSimpleState, createFinalState, createInitialState, createTransition, createEvent, and createAction in layer 1.

Layer 2 consists of rules for link creation for associations with multiplicity [1,1] at one association end. The rules have to be applied as long as possible. We have rules that create links between existing instances and rules that create an instance (of a concrete type) and a link to this instance starting at an instance that is not yet connected to another instance. New instances can only be created if there are not enough instances in the graph what is ensured by (negative) application conditions. For the association source between StateVertex and Transition, we derive four rules: one rule creates a link source between an existing StateVertex and an existing Transition. Further,



Fig. 3. Example Grammar Rules 2

for each concrete class that inherits from class StateVertex one rule is derived that creates the StateVertex, an InitialState, a CompositeState, SimpleState or a FinalState, and the link source. Note that the abstract class StateVertex could be matched to any of its concrete subclasses InitialState, CompositeState, FinalState, and SimpleState. For the association target between StateVertex and Transition, similar rules are derived. For the association top between StateMachine and CompositeState, we derive two rules. One of them is shown in Figure 3, creating a CompositeState to a StateMachine if no CompositeState exists in the instance graph.

Layer 3 consists of rules creating links for associations with multiplicity [0,1] or [0,\*] at the association ends. The graph grammar derivation rules in layer 3 can be applied *arbitrarily often*. The rules in layer 3 are terminating. But in order to generate all possible instances, the rule application can be interrupted by user interaction or after a random time period. The rules create links between existing instances, so they have negative application conditions prohibiting the insertion of more links than allowed by the meta model cardinalities. For the running example, the rules of layer 3 are shown in Figure 4.

We further get rules that insert links for the association between Transition and Action and the association between Transition and Event as well as association between CompositeState and StateVertex.

The example rules shown in Figures 2 - 4 construct a simple instance graph consisting of a StateMachine with its top CompositeState containing three state vertices and two transitions between them. In the application



Fig. 4. Example Grammar Rules 3

conditions shown in Figures 2 - 4 the node types are abbreviated (CS for CompositeState etc.).

# 4 Translation of Restricted OCL Constraints into Graph Constraints

Up to now there is no general way to transform OCL constraints into equivalent graph constraints, which are introduced in [EEHP04]. As a first approach, we show how restricted OCL constraints can be translated to equivalent graph constraints. In contrast to the translation of meta models to graph grammars which was described formally in the previous chapters, we discuss first ideas for the translation of OCL constraints only and sketch how they can be ensured. Besides having one common formalism the motivations for translating OCL constraints into graph constraints is their later consideration within the derivation process (sketched below).

We restrict OCL constraints to equality, size, and attribute operations for navigation expressions, called *restricted OCL constraints*. In future work, OCL constraints and graph constraints have to be further compared concerning their expressiveness.

Graph constraints are properties on graphs which have to be fulfilled. They are used to express contextual conditions like the existence or non-existence of certain nodes and edges or certain subgraphs in a given graph. Application conditions for rules were first introduced in [EEHP04]. They restrict the capability of rules, e.g. a rule can be applied if certain nodes and edges or certain subgraphs in the given graph exist or do not exist.

**Definition 4.1** [graph constraint] *Graph constraints* over an object P are defined inductively as follows: For a graph morphism  $x : P \to C$ ,  $\exists x$  is a (*basic*) graph constraint over P. For a graph morphism  $x : P \to C$  and a graph



Fig. 5. All Transitions have exactly n source [target] vertices

 $\forall \left[ \underbrace{1:\text{Transition}}_{1:\text{Transition}} \rightarrow \underbrace{1:\text{Transition}}_{1:\text{Transition}} \right], \quad (\bigvee_{n \in N^+} (c_{\text{source-n}} \land c_{\text{target=n}})) \right]$ 

Fig. 6. All Transitions have the same number of source and target vertices

constraint c over C,  $\forall (x, c)$  and  $\exists (x, c)$  are (conditional) graph constraints over P. For graph constraints  $c, c_i (i \in I)$  [over P], true, false,  $\neg c, \land_{i \in I} c_i$  and  $\lor_{i \in I} c_i$ are (Boolean) graph constraints [over P].

A graph morphism  $p: P \to G$  satisfies a basic graph constraint  $\exists x$  if there exists a graph morphism  $q: C \to G$  with  $q \circ x = p$ . A graph morphism  $p: P \to G$  satisfies a conditional graph constraint  $\forall (x,c) \ [\exists (x,c)]$  if for all [some] graph morphisms  $q: C \to G$  with  $q \circ x = p$ , q satisfies c. A graph morphism p satisfies a Boolean graph constraint  $\neg c$  if p does not satisfy c; psatisfies  $\lor_{i \in I} c_i \ [\land_{i \in I} c_i]$  if p satisfies all [some]  $c_i$  with  $i \in I$ .

A graph morphism  $p: P \to G$  satisfies a basic graph constraint  $\exists x$  if there exists a graph morphism  $q: C \to G$  with  $q \circ x = p$ . A graph morphism  $p: P \to G$  satisfies a conditional graph constraint  $\forall (x,c) \ [\exists (x,c)]$  if for all [some] graph morphisms  $q: C \to G$  with  $q \circ x = p$ , q satisfies c. A graph morphism p satisfies a Boolean graph constraint  $\neg c$  if p does not satisfy c; psatisfies  $\lor_{i \in I} c_i \ [\land_{i \in I} c_i]$  if p satisfies all [some]  $c_i$  with  $i \in I$ .

A graph *G* satisfies a graph constraint *c* of the form  $\exists x, \exists (x, d) \ [\forall (x, d)]$  if all [some] graph morphisms  $p: P \to G$  satisfy *c*. A graph *G* satisfies  $\neg c$  if *G* does not satisfy *c* and  $\bigvee_{i \in I} c_i \ [\wedge_{i \in I} c_i]$  if it satisfies all [some]  $c_i$  with  $i \in I$ .

With this definition of graph constraints the counting of elements is possible. For the state chart example we can express graph properties like: "All Transitions have exactly n source [target] vertices" (Figure 5), or "All Transitions have the same number of source and target vertices" (Figure 6). Therefore we define Boolean graph constraints  $t_{source=n}$  and  $t_{target=n}$  expressing that the Transition has n source [target] vertices and not n + 1 source [target] vertices, where  $S_n[T_n]$  denotes the star with n sources [targets]. In the conditional graph constraint in Figure 6 we need the basic graph constraint that maps only a Transition node to a Transition node, since the Transition node in  $S_n$  has to be the same as in  $T_n$ .

The restricted OCL constraints that can be translated are divided into atomic navigation expressions and complex navigation expressions.



Fig. 7. Examples for Translation of OCL Constraints

#### Atomic navigation expressions:

Atomic navigation expressions are OCL expressions that

- express equivalent navigations,
- end with operation *size()* (if the result is compared with constants),
- end with operations *isEmpty()*, *notEmpty()* or *isUnique()*, or
- end with attribute operations (not considered explicitly in the paper).

The navigation expressions contain navigation along association ends or association classes only.

Atomic navigation expressions can be transformed into basic graph constraints of the form  $\exists x$  or boolean formulae over basic graph constraints.

A navigation expression stating that two navigations have the same result, like self.ass1=self.ass2.ass3, can be transformed into a graph constraint, see Figure 7 a). Here the conclusion of the constraint ensures that ass1 and ass3 are connected to the same instance of Class2.

Operation size() can be translated into a Boolean graph constraint that is composed of two basic graph constraints, see Figure 7 b). The first constraint ensures that there exist the minimum number (= value of the constant) of association ends, the second prohibits the existence of more than the constant value association ends. If the comparison operation is  $\leq$  or  $\geq$  the OCL constraint can be translated into just one graph constraint.

Operations isEmpty() and notEmpty() can be translated back to a size() operation: self.ass1->isEmpty() is translated back to self.ass1->size()=0, self.ass1->notEmpty() to self.ass1->size()>=1.

Collection operation isUnique() can be translated into a size() operation, if the body of the collection operation is a navigation expression ending at an instance set: self.ass1->isUnique(navexp) is translated back to

self.ass1.navexp->size()<=1.</pre>

#### Complex navigation expressions:

**Definition 4.2** [complex navigation expressions] Atomic navigation expressions are complex navigation expressions. Given complex navigation expressions a, b and c, expressions not(a), a and b, a or b, a implies b, and if a then b else c are complex navigation expressions.

Complex navigation expressions can be transformed into conditional graph constraints as described in the following.

An OCL expression of the form a implies b is equivalent to the expression  $\operatorname{\mathsf{not}}(a)$  or b. So we have to translate  $\operatorname{\mathsf{not}}(a)$  or b into an equivalent conditional graph constraint. First the expressions a and b are transformed into graph constraints  $c_a$  and  $c_b$  as described above. We have to combine the two graph constraints  $c_a$  and  $c_b$  by the operator  $\wedge$ , and therefore we have to find the part that has to be identified in both expressions. So we build the intersection expression of the premise and the conclusion, that is a navigation part contained in both expressions (in the example constraint for Transition in Figure 8, the node of type Transition is in this intersection only). This intersection expression *ie* occurs in the left graph of the graph constraints  $c_a$  and  $c_b$ . Having  $c_a$ ,  $c_b$  and *ie*, we can build a conditional graph constraint that is equivalent to the OCL constraints as follows: Build the basic graph constraint  $b : ie \to ie$ . Build the conditional graph constraint  $\exists (b, \neg (c_a) \lor (c_b))$ , where the intersection expression is mapped to the corresponding elements in  $c_a$  and  $c_b$ . See the description of Figure 8 for an example.

OCL constraints of the form if a then b else c can be translated back into two implies operations: (a implies b) and ((not a) implies c). The implies expressions are translated as described before into two graph constraints which then are combined by the logical operator ( $\wedge$ ) to a new one that is equivalent to the OCL constraint.

#### Ensuring of graph constraints:

Ensuring of graph constraints can be done in two ways: One is to check constraints once the overall derivation of an instance model has terminated which would also be the approach followed when checking OCL constraints directly. However, this leads to the generation of a large number of non-valid instances in between. A more promising approach is to take the constraints into consideration during the derivation process: For each class in the meta model the corresponding graph constraints can be identified. For rules of layer 1, constraints are ignored. For rules of layer 2 and 3, negative constraints of the form  $\neg \exists x, \neg \exists (x, c), \neg \forall (x, c)$ , where x is a basic graph constraint and c is a graph constraint, for the participating classes are evaluated before a possible application of a rule. If the resulting instance violates a constraint, the previous application of a rule is not executed. Here we use the translation of



Fig. 8. Translation of OCL Constraints for Statechart Meta Model

graph constraints to application conditions as presented in [EEHP04]. Positive constraints of the form  $\exists x, \exists (x,c), \forall (x,c)$  are checked after termination of layer 3. If a positive constraint is violated, the model can be fixed by adding additional elements required by the positive constraint. It remains to future work to determine those negative constraints that can be violated by adding the elements required by a positive constraint and to extend the formalization in the previous sections to constraints.

#### Translation of the OCL constraints for the statechart meta model:

Figure 8 shows the translation of the OCL constraints for the simple statechart meta model example in Figure 1. The first translates the OCL constraint context FinalState inv: self.incoming->size()>=1 (that is an atomic navigation expression) into an equivalent basic graph constraint. This constraint corresponds to the size()-operation constraint shown in Figure 7 c). The second translates the OCL constraint context FinalState inv: self.outgoing->size()=0 into an equivalent basic graph constraint, corresponding to the graph constraints shown in Figure 7 b). Note, that the positive graph constraint is not needed if size() is compared to 0. The third one is similar. The OCL constraint context Transition inv: self.source.ocllsTypeOf(InitialState) implies self.target.ocllsKindOf(State) is a complex navigation expression. It is equiva-(not(self.source.ocllsTypeOf(InitialState))) lent to the expression or (self.target.ocllsKindOf(State)), stating that each source instance of a Transition instance is not an InitialState or the target instance is a State. The two OCL expressions can be translated into two basic graph constraints shown in Figure 8 (the lower part of the last graph constraint). We have to combine the two basic graph constraints by operator  $\vee$  and we have to express that the Transition instance in one expression is the same as in the other expression. Therefore we have to build the intersection expression of the premise and

the conclusion, which contains the Transition only. The complete conditional graph constraint states: all nodes of type Transition have a source node of type InitialState or a target node of type State. This is equivalent to: *Transitions outgoing InitialStates must always target a State*.

# 5 Related Work

One of the related problems is the one of automated snapshot generation for class diagrams for validation and testing purposes, tackled by Gogolla et al. [GBR05]. In their approach, properties that the snapshot has to fulfill are specified in OCL. For each class and association, object and link generation procedures are specified using the language ASSL. In order to fulfill constraints and invariants, ASSL offers try and select commands which allow the search for an appropriate object and backtracking if constraints are not fulfilled. The overall approach allows snapshot generation taking into account invariants but also requires the explicit encoding of constraints in generation commands. As such, the problem tackled by automatic snapshot generation is different from the meta model to graph grammar translation.

Formal methods such as Alloy [All00] can also be used for instance generation: After translating a class diagram to Alloy one can use the instance generation within Alloy to generate an instance or to show that no instances exist. This instance generation relies on the use of SAT solvers and can also enumerate all possible instances. In contrast to such an approach, our approach aims at the construction of a grammar for the metamodel and thus establishes a bridge between metamodel-based and grammar-based definition of visual languages.

# 6 Conclusion and Future Work

In this paper we have presented the main concepts for translating a meta model to an instance generating graph grammar. This translation has been illustrated by a simple statecharts example. The meta classes and associations as well as their multiplicities are translated directly to type graph, start graph and graph rules. To handle also the OCL constraints during the instance generation process, they are first translated to graph constraints and then partly to application conditions of rules. In this paper, we discussed this translation process for restricted OCL constraints. In future work, OCL constraints and graph constraints have to be further compared concerning their expressiveness. Moreover, we started to give OCL a new kind of semantics which has to be set into relation with other OCL semantics.

Automatic derivation of instances from meta models is a complex task which needs tool support. So far, we have automated the construction of an instance generating graph grammar by providing a model transformation that automatically derives an instance generating graph grammar from a meta model. For a complete description of this implementation we refer to the URL http://tfs.cs.tu-berlin.de/agg/MM2GraGra. Although the current model transformation does not support all features of meta models yet, it nevertheless shows the feasibility of our approach. We plan to investigate the usage of our techniques for systematic testing of model transformations at different development stages of our generation algorithm.

### References

[All00] The Alloy Analyzer - 3.0 Beta http://alloy.mit.edu/, 2000.

- [BEdLT04] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, Proc. Fundamental Aspects of Software Engineering 2004, volume 2984. Springer LNCS, 2004.
  - [BS97] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.
- [EEdL<sup>+</sup>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.
- [EEHP04] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), LNCS 3256, pages 287–303, Rome, Italy, October 2004. Springer.
- [EKTW05] K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Automatically Generating Instances of Meta Models based on a Graph Grammar Approach. Technical Report 2005–09, Technical University of Berlin, Dept. of Computer Science, November 2005. To appear.
  - [GBR05] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. Software and Systems Modeling, 2005. To appear.
  - [Obj05] Object Management Group (OMG). OCL 2.0 Specification. OMG document ptc/2005-06-06, June 2005.
  - [UML03] Object Management Group (OMG). UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02, August 2003.