OCL2AC

Automatic Translation of OCL Constraints to Graph Constraints and Application Conditions for Transformation Rules

Nebras Nassar[1](✉)[0000−0002−0838−6513], Jens Kosiol[1][0000−0003−4733−2777], Thorsten Arendt[2][0000−0002−4866−6405], and Gabriele Taentzer[1][0000−0002−3975−5238]

[1] Philipps-Universität Marburg, Marburg, Germany
{nassarn,kosiolje,taentzer}@informatik.uni-marburg.de
[2] GFFT Innovationsförderung GmbH, Bad Vilbel, Germany
thorsten.arendt@gfft-ev.de

Abstract. Based on an existing theory, we present a tool *OCL2AC* which is able to adapt a given rule-based model transformation such that resulting models guarantee a given constraint set. *OCL2AC* has two main functionalities: First, OCL constraints are translated into semantically equivalent graph constraints. Secondly, graph constraints can further be integrated as application conditions into transformation rules. The resulting rule is applicable only if its application does not violate the original constraints. *OCL2AC* is implemented as Eclipse plug-in and enhances Henshin transformation rules.

Keywords: OCL · Nested Graph Constraints · Model Transformation · Henshin

1 Introduction

Model transformations are the heart and soul of Model-Driven Engineering (MDE). They are used for various MDE-activities including translation, optimization, and synchronization of models [13]. Resulting models should belong to the transformation's target language which means that they have to satisfy all the corresponding language constraints. Consequently, the developer has to design transformations such that they behave well w.r.t. language constraints.

Based on existing theory [8,12], we developed a tool, called *OCL2AC*, which automatically adapts a given rule-based model transformation such that resulting models satisfy a given set of constraints. Use-cases for this tool are abundant, including instance generation [12], ensuring that refactored models do not show certain model smells (anymore), and generating model editing rules from meta-models to enable high-level model version management [9].

Our tool builds upon the following basis: The de facto standard for defining modeling languages in practice are the Eclipse Modeling Framework (EMF) [5] for specifying meta-models and the Object Constraint Language (OCL) [10] for expressing additional constraints. Graph transformation [6] has been shown to be a versatile foundation for rule-based model transformation [7] focusing on the models' underlying graph structure. To reason about graph properties, Habel and Pennemann [8] have developed (nested) graph constraints being equivalent to first-order formulas on graphs.

OCL2AC consists of two main components: The first component *OCL2GC* translates a reasonable subset of OCL constraints to graph constraints using the formally defined OCL translation in [12] as conceptual basis. The second component *GC2AC* integrates graph constraints as application conditions into transformation rules specified in Henshin, a language and tool environment for EMF model transformation [2]. The resulting application conditions ensure that EMF models resulting from applications of the enhanced rules do not violate the original constraints. The rules' actions are not changed by the integration. Each of these two components is designed to be usable on its own.

Note that our OCL translator is novel: Instead of checking satisfiability, it enhances transformation rules such that their applications can result in valid models only. To that extent, *OCL2AC* can be used to not just check constraint satisfaction but also to tell the user how to improve transformation rules.

The paper is structured as follows: In Sect. 2, we present preliminaries. Section 3 presents the two main components of the tool and their internal functionalities. Related work is given in Sect. 4 while Sect. 5 concludes the paper.

2 Preliminaries

2.1 Introductory Example

To illustrate the behaviour of our tool, we use a simple Statecharts meta-model displayed in Fig. 1.

A StateMachine contains at least one Region which may potentially contain Transitions and Vertices. Vertex is an abstract class with a concrete subclass State. FinalState inherits from State. A State can again contain Regions. Transitions connect Vertices.

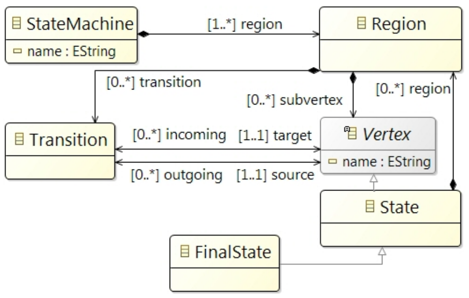

Fig. 1. A simple Statecharts meta-model

A basic constraint on Statecharts which is not expressible by just the graphical structure of the meta-model or by multiplicities is: A FinalState has no outgoing transition. We name this constraint no_outgoing_transitions.

2.2 OCL

The Object Constraint Language (OCL) [10] is a constraint language used to supplement the specification of object-oriented models. OCL constraints may be used to specify invariants, operation contracts, or queries. The constraint no_outgoing_transitions can be specified in OCL as:

```
context FinalState invariant no_outgoing_transitions:
  self.outgoing->isEmpty();
```

Our technique supports a slightly restricted subset of Essential OCL [10]. Since OCL constraints are translated to nested graph constraints and thereby get a precise semantics, we focus on OCL constraints corresponding to a first-order, two-valued logic and relying on sets as the only collection type. Also, there is only limited support for user-defined operations. Details can be found in [1,12].

2.3 Graph Rules, Graph Conditions, and Graph Constraints

Our tool currently enhances Henshin rules [2]. A rule specifies elements as to be deleted, created, or preserved at application. Additionally, it may be equipped with an *application condition* controlling its applicability. Figure 2 shows a Henshin rule insert_outgoing_transition. When applying it at chosen nodes Vertex and Transition in an instance, an edge of type outgoing is created between them.

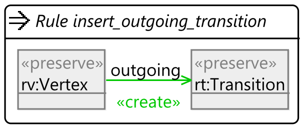

Fig. 2. Transformation rule

$$\forall\left(\boxed{\text{self:FinalState}},\ \nexists\ \boxed{\text{self:FinalState}} \xrightarrow{\text{outgoing}} \boxed{\text{var9:Transition}}\right)$$

Fig. 3. Graph constraint no_outgoing_transitions

(Nested) graph constraints are invariants which are checked for all graphs, whereas (nested) graph conditions express properties of morphisms. The primary example for that are application conditions for transformation rules. Constraints are special cases of conditions since the empty graph can be included into any graph. A version, important from the practical point of view, are *compact constraints and conditions* [12]. They allow for dense representation and obtain their semantics by *completing* them into nested constraints or conditions. All those formalisms allow to express first-order properties [8]. As an example, the graph constraint in Fig. 3 states that a FinalState does not have an outgoing transition.

3 Tooling and Architecture

We implemented an Eclipse plug-in, called *OCL2AC*, with two components: (1) *OCL2GC* takes an Ecore meta-model and a set of OCL constraints as inputs and automatically returns a set of semantically equivalent graph constraints as output. (2) *GC2AC* takes a transformation rule defined in Henshin and a graph constraint, possibly compact, as inputs, and automatically returns the Henshin rule with an updated application condition guaranteeing the given graph constraint. Each component can be used independently as an Eclipse-based tool. The tool is available for download at [1]. We introduce the architecture and internal processes of both components and highlight some additional features.

3.1 From OCL to Graph Constraints

The first component of our tool takes an Ecore meta-model and a set of OCL constraints as inputs and returns a set of semantically equivalent (nested) graph constraints as output. The translation process is composed of the steps shown in Fig. 4, which can be automatically performed.

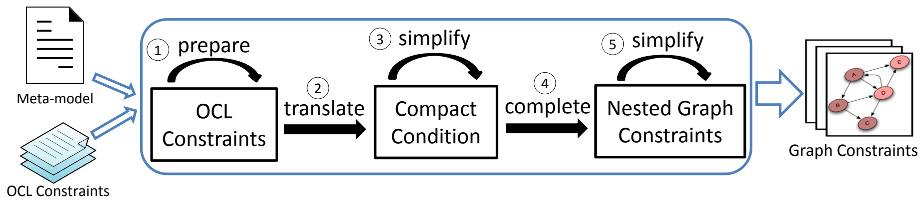

Fig. 4. From OCL to graph constraints: Component design

In Step (1) an OCL constraint is prepared by refactorings. This is done to ease the translation process, especially to save translation rules for OCL constraints. The semantics of the constraint is preserved during this preparation. Step (2) translates an OCL constraint to a graph constraint along the abstract syntax structure of OCL expressions. This translation largely follows the one in [12]. Let us consider the translation of OCL constraint no_outgoing_transitions to the graph constraint displayed in Fig. 3: The expression self.outgoing→isEmpty() is refactored to not(self.outgoing→size() $\geq$ 1). Hence, a translation rule for isEmpty() is not needed. Then, this sub-expression is translated to a compact condition containing a graph with one edge of type outgoing from a node of type FinalState to a node of type Transition. The existence of such a pattern is negated.

Then a first simplification of the resulting compact condition takes place in Step (3), using equivalence rules [11,12]. Applying those can greatly simplify the representation of a condition; they can even collapse nesting levels.

Step (4) completes the compact condition to a nested graph constraint being used to compute application conditions later on. The resulting nested graph constraint is simplified in Step (5) using again equivalence rules. Our tool allows to display the resulting constraint as nested graph constraint or as compact constraint. The intermediate steps are computed internally only.

3.2 From Graph Constraints to Left Application Conditions

The second component of our tool takes a Henshin rule and a graph constraint as inputs, and returns the Henshin rule with an updated application condition guaranteeing the given graph constraint. Figure 5 gives an overview of the steps to be performed:

In Step (1) the given graph constraint is prepared; if the input is a compact constraint, it is expanded to a nested constraint. Moreover, it is refactored to eliminate syntactic sugar. The operator $\Rightarrow$ for implication, for example, is replaced with basic logic operators. Step (2) shifts a given graph constraint gc to

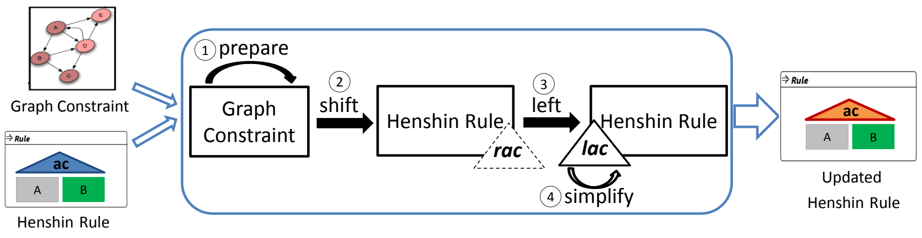

Fig. 5. Integration as application conditions: Component design

the RHS of the given rule so that we get a new right application condition *rac* for this rule. The main idea of shift is to consider all possible ways in which *gc* could be satisfied after rule application. This is done by overlapping the elements of the rule's RHS with each graph of *gc* in every possible way. This overlapping is done iteratively along the nesting structure of *gc*. This algorithm is formally defined in [8] and shown to be correct. The result of this calculation is yet impractical as one would need to first apply the rule and then check the right application condition to be fulfilled. Therefore, we continue with the next step.

Step (3) translates a right application condition *rac* to the LHS of the given rule *r* to get a new left application condition *lac*. It is translated by applying the rule reversely to the right application condition *rac*, again along its nesting structure. If the inverse rule of *r* is applicable, the resulting condition is the new left application condition. Otherwise *r* gets equipped with the application condition `false`, as it is not possible to apply *r* at any instance without violating *gc*. The rule *r* with its new application condition has the property that, if it is applicable, the resulting instance fulfills the integrated constraint. Step (4) simplifies the resulting left application condition using equivalences as for graph conditions. The output is the original Henshin rule with an updated left application condition guaranteeing the given graph constraint.

For example, integrating the constraint no_outgoing_transitions into the rule insert_outgoing_transition results in the application condition displayed in Fig. 6. The upper part forbids node rv:Vertex being matched to a FinalState. The lower part requires that the rule is matched to consistent models only, i.e., not containing already a FinalState with an outgoing Transition. It may be omitted if consistent input models can always be assumed.

$$\neg\exists\left(\begin{array}{c}\boxed{\text{rv:FinalState}}\\ \\ \boxed{\text{rt:Transition}}\end{array}, \exists\begin{array}{c}\boxed{\text{rv:FinalState}}\xrightarrow{\text{outgoing}}\boxed{\text{var9:Transition}}\\ \\ \boxed{\text{rt:Transition}}\end{array} \vee \exists\begin{array}{c}\boxed{\text{rv:FinalState}}\\ \\ \boxed{\text{rt:Transition}}\end{array}\right)\wedge$$

$$\neg\exists\left(\begin{array}{c}\boxed{\text{rv:Vertex}}\\ \boxed{\text{rt:Transition}}\\ \boxed{\text{self:FinalState}}\end{array}, \exists\begin{array}{c}\boxed{\text{rv:Vertex}}\\ \boxed{\text{rt:Transition}}\\ \boxed{\text{self:FinalState}}\xrightarrow{\text{outgoing}}\boxed{\text{var9:Transition}}\end{array} \vee \exists\begin{array}{c}\boxed{\text{rv:Vertex}}\\ \\ \boxed{\text{self:FinalState}}\xrightarrow{\text{outgoing}}\boxed{\text{rt:Transition}}\end{array}\right)$$

Fig. 6. Application condition for the rule insert_outgoing_transition after integrating the constraint no_outgoing_transitions

Tool Features. *OCL2AC* provides a wizard for selecting a rule and a graph constraint that shall be integrated. The inputs of the wizard are a Henshin model (file) and a graph constraint model being generated by *OCL2GC* or manually designed based on the meta-model for compact conditions (at [1]). *OCL2AC* additionally provides tool support for *pretty printing graph constraints and application conditions* of Henshin rules in a graphical form as shown in Figs. 3 and 6 above. Pretty printing is supported for both compact and detailed representation of nested constraints and application conditions. The output of the pretty printing is a LaTeX-file being rendered as pdf-file and displayed in a developed Eclipse PDF viewer.

4 Related Work

We briefly compare our tool with the most related tools for translating OCL or calculating application conditions. To the best of our knowledge, we present the first ready-to-use tool for integrating constraints as application conditions into transformation rules.

In [3], Bergmann proposes a translation of OCL constraints into graph patterns. The correctness of that translation is not shown. The implementation covers most of that translation. In [11], Pennemann introduces ENFORCe which can check and ensure the correctness of high-level graph programs. It integrates graph constraints as left application conditions of rules as well. However, the tool is not published to try that out. Furthermore, there is no translation from OCL to graph constraints available. Clarisó et al. present in [4] how to calculate an application condition for a rule and an OCL constraint, directly in OCL. They provide a correctness proof and a partial implementation.

5 Conclusion

OCL2AC automatically translates OCL constraints into semantically equivalent graph constraints and thereafter, it takes a graph constraint and a Henshin rule as inputs and updates the application condition of that rule such that it becomes constraint-guaranteeing. *OCL2AC* is a ready-to-use tool implemented as Eclipse plug-in based on EMF and Henshin. As future works, we intend to use it for improving transformation rules for various modeling purposes such as model validation and repair.

Acknowledgement. We are grateful to Jan Steffen Becker, Annegret Habel and Christian Sandmann for their helpful support. This work was partially funded by the German Research Foundation (DFG), projects "Generating Development Environments for Modeling Languages" and "Triple Graph Grammars (TGG) 2.0".

References

1. OCL2AC: Additional material, https://ocl2ac.github.io/home/, (2018)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Proc. of MODELS 2010. pp. 121–135. Springer (2010). https://doi.org/10.1007/978-3-642-16145-2
3. Bergmann, G.: Translating OCL to Graph Patterns. In: Proc. of MODELS 2014. pp. 670–686. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_41
4. Clarisó, R., Cabot, J., Guerra, E., de Lara, J.: Backwards reasoning for model transformations: Method and applications. Journal of Systems and Software **116**(Supplement C), 113–132 (2016). https://doi.org/https://doi.org/10.1016/j.jss.2015.08.017
5. Eclipse Foundation: Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/, (2018)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer (2006)
7. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation – General Framework and Applications. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2015)
8. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19**, 245–296 (2009)
9. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: Proc. of ICMT 2016. pp. 173–188. Springer (2016). https://doi.org/10.1007/978-3-319-42064-6_12
10. OMG: Object Constraint Language, http://www.omg.org/spec/OCL/
11. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Carl von Ossietzky-Universität Oldenburg (2009)
12. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. Science of Computer Programming **152**, 38 – 62 (2018)
13. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20**(5), 42–45 (2003)