Editing Nested Constraints and Application Conditions

Karl Azab

Carl v. Ossietzky Universität Oldenburg, Germany *
azab@informatik.uni-oldenburg.de

Abstract. Nested constraints and application conditions express first-order properties on graphs and are implemented in the system ENFORCe. While ENFORCe has been without a GUI, ROOTS is a new GUI for the graph transformation engine of AGG and provides standard editing functionality for graphs and graph transformation rules. We integrated ENFORCe with ROOTS to get these editing features and implemented a tree-oriented visualization for editing nested constrains and application conditions.

1 Introduction

Graph programs use graph transformations to compute relations on graphs [1]. Graph transformations is a formalism which describes how graphs are rewritten by rules with graphs as left and right hand sides. An overview of the computations that can be made by graph transformations is given in [2–4].

We use nested constraints and applications conditions – (nested) graph conditions for short – as described in e.g. [5] as a visual representation of first-order logic formulas on graphs. As constraints, graph conditions express properties on graphs and as application conditions they limit the applicability of graph transformation rules. Given a (double-pushout) rule, there are transformations of constraints into application conditions that limit the applicability of that rule in such a way that the constraint is satisfied if the rule can be applied [6]. When graph conditions specify pre- and postconditions on graph programs, a weakest precondition can be computed [7].

ENFORCe [8] (ENsuring FORmal Correctness of high-level programs) implements the transformations and static analysis methods for nested graph conditions mentioned above. Transformations are done on a categorical level with the help from structure specific plug-ins (also called *engines*). ENFORCe has an engine for directed labeled graphs but input and output from ENFORCe has been via text files which need to be manually created and interpreted. This is of course

* This work is supported by the German Research Foundation (DFG) under grant no. HA 2936/2 (Development of Correct Graph Transformation Systems). The author thanks Stefan Jurack for making the ROOTS software available and providing technical assistance. We also thank Annegret Habel and Karl-Heinz Pennemann for commenting on a draft of this paper.

a tedious and error prone method. To more conveniently access ENFORCe, a GUI for graphs, rules, and nested graph conditions is needed.

ROOTS [9] (Rule based Object Oriented Transformation System) is a new GUI for the graph transformation engine of AGG [10]. It contains editors for e.g. graphs and rules and supports graph layout. ROOTS is based on an extensible EMF model which is only loosely coupled to AGG and can therefore relatively easily be ported to other graph transformation tools. While ROOTS have editors for atomic constraints and negative application conditions, it does not natively support nested constraints and application conditions.

There are other GUIs for graph transformation systems, e.g. Fujaba [11], with established plugin architectures, but both ROOTS/AGG and ENFORCe work on double-pushout graph transformations and their verification techniques. In a longer perspective, we hope this similarity will have a synergetic effect on the integration's usability.

This paper describes the integration of ENFORCe and ROOTS and the implemented extensions necessary for editing nested graph conditions. The integration was done at the interface between ROOTS and AGG: synchronization routines were added for the features in ROOTS that are also available in ENFORCe. The new editor extensions for graph conditions are synchronized to ENFORCe's internal data structures, see the overview in Fig. 1.

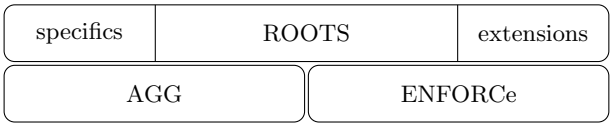

Fig. 1. Overview of the integration.

The paper is organized as follows: In Sect. 2 we review graph conditions and explain the visualization approach we used for nested graph conditions. Section 3 describes how ENFORCe was integrated with ROOTS, the implemented extensions to ROOTS and show how existing view components could be reused with only small changes. We conclude our work in Sect. 4.

2 Layout of Graph Conditions

In this section we first review the formal definition of graph conditions and then show the basic idea of how the implemented editor visualizes them. We use standard definitions of graphs and graph morphisms – as in e.g. [12].

Definition 1 (graphs). A *label alphabet* $\mathrm{C} = \langle \mathrm{C_V}, \mathrm{C_E} \rangle$ consists of two finite sets of node and edge labels. A *graph* over C is a six-tuple $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_G, \mathrm{m}_G)$ consisting of two finite sets V_G and E_G of *nodes* and *edges*, a *source* and a *target*

function for edges $\mathrm{s}_G, \mathrm{t}_G \colon \mathrm{E}_G \to \mathrm{V}_G$, and two *labeling functions* $\mathrm{l}_G \colon \mathrm{V}_G \to \mathrm{C}_\mathrm{V}$ and $\mathrm{m}_G \colon \mathrm{E}_G \to \mathrm{C}_\mathrm{E}$. A graph with an empty set of nodes is *empty* and denoted by $\emptyset$. The set of all graphs over C is denoted by $\mathcal{G}_\mathrm{C}$. An a-node (edge) in G is an element in V_G (E_G) with label a.

Example 1. Let us model two responsibilities of an operating system: scheduling and resource allocation. A core, process, and resource is represented by a node labeled C, P, and R, respectively. That a core is devoted to executing the threads of a particular process is modeled by an edge from a C-node to a P-node. A request from a process to obtain a resource is modeled by an edge from a P-node to an R-node. That a resource has been assigned to a process is modeled by an edge from an R-node to a P-node. These relationships are depicted in Fig. 2.

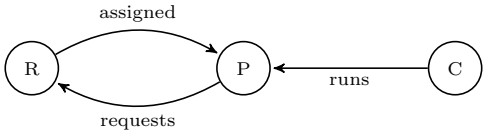

Fig. 2. Type graph of the operating system example.

Nested graph conditions contain graph morphisms.

Definition 2 (graph morphisms). A *(graph) morphism* $g \colon G \to H$ consists of two total functions $g_\mathrm{V} \colon \mathrm{V}_G \to \mathrm{V}_H$ and $g_\mathrm{E} \colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve sources, targets, and labels, that is, $\mathrm{s}_H \circ g_\mathrm{E} = g_\mathrm{V} \circ \mathrm{s}_G$, $\mathrm{t}_H \circ g_\mathrm{E} = g_\mathrm{V} \circ \mathrm{t}_G$, $\mathrm{l}_H \circ g_\mathrm{V} = \mathrm{l}_G$, and $\mathrm{m}_H \circ g_\mathrm{E} = \mathrm{m}_G$. It is *injective* if g_V and g_E are injective. The *composition* $h \circ g$ of g with a morphism $h \colon H \to M$ consists of the composed functions $h_\mathrm{V} \circ g_\mathrm{V}$ and $h_\mathrm{E} \circ g_\mathrm{E}$.

We define graph conditions as in [5–7].

Definition 3 (nested graph conditions). A *graph condition* over a graph P is of the form $\exists a$ or $\exists(a, c)$, where $a \colon P \to C$ is an injective graph morphism and c is a condition over the graph C. Moreover, Boolean formulas over conditions (over P) are conditions (over P). A morphism $p \colon P \to G$ *satisfies* a condition $\exists a$ ($\exists(a, c)$) over P if there exists an injective morphism $q \colon C \to G$ with $q \circ a = p$ (satisfying c). A graph G *satisfies* a condition $\exists a$ ($\exists(a, c)$) if all injective morphisms $p \colon P \to G$ satisfy the condition. The satisfaction of conditions over P by graphs or morphisms with domain P is extended to Boolean formulas over conditions in the usual way. We use an abbreviation $\forall(a, c)$ to denote $\neg\exists(a, \neg c)$.

Since we are interested in editing, we will focus on the syntactic structure of graph conditions, the reader interested in the semantics of nested graph conditions is referred to e.g. [5]. We use graph conditions in different contexts – on graphs, then called constraints and graph transformation rules, then referred to as application conditions, see e.g. [5, 6] for details.

Example 2. We can formulate two desirable properties for the operating system model from Example 1: (1) *A resource may at most be assigned to one process*, which expressed as a graph condition looks like Fig. 3 (a) and (2) *Every core is assigned to run threads in exactly one process*, shown in Fig. 3 (b).

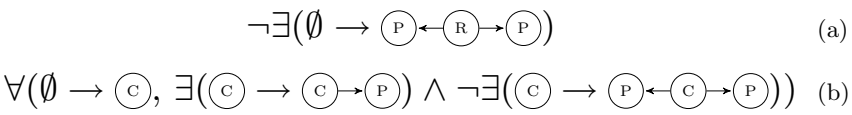

Fig. 3. (a) A graph condition for resource allocation and (b) a nested graph condition for scheduling.

As already becoming apparent from the condition in Fig. 3, large graph conditions can be difficult to read. To present a survey of a graph condition, we represent graph conditions as trees: A quantifier together with its morphism becomes a node and its nested condition becomes its child. Boolean formulas are represented as trees in the obvious way. To simplify the layout and reuse existing view components, we show the graph condition tree in two views: one for the tree structure and one for morphisms. We then only need to layout trees and morphisms separately – and tree layout is easy and for morphisms the only difficult part is to layout the domain and codomain, and functionality for graph layout already exists in ROOTS. The tree layout for the nested graph condition from Fig. 3 is shown in Fig. 4 (a). The separated tree and morphism view of that graph condition is shown in Fig. 4 (b). In the latter figure, a, b_1 and b_2 identifies the entry point of the morphism view into the tree view.

3 Implementation

ROOTS is implemented as an Eclipse plugin and is based on the model-view-controller pattern, where the model is generated by EMF [13]. That generated model is supplemented with routines to synchronize it with AGG's internal data structures. Our integration adds synchronization routines to ENFORCe for the common functionality of AGG and ENFORCe. The extensions for graph conditions are only synchronized to ENFORCe, as there is no suitable data structure available in AGG. In this way, many editing features become (for the end-user) seamlessly available also for ENFORCe, see Fig. 5. An advantage of using an ap-

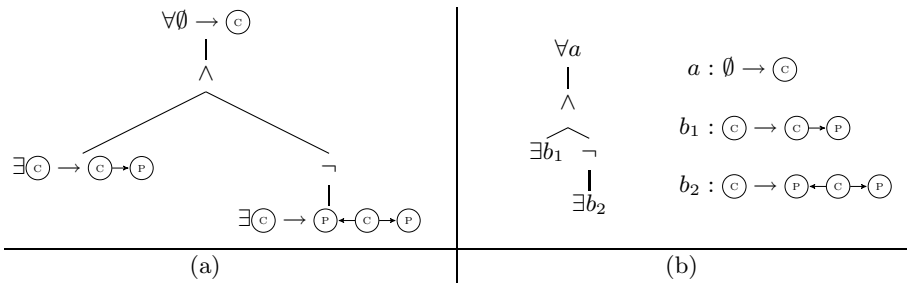

Fig. 4. (a) Tree representation and (b) two views of of a condition.

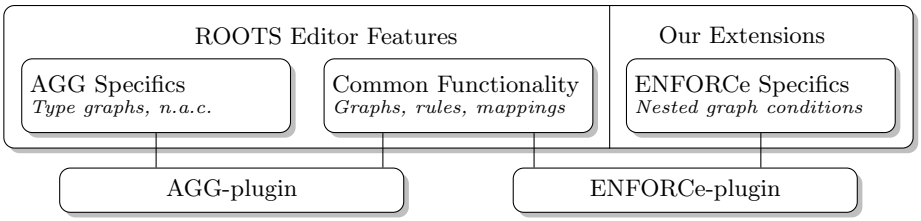

Fig. 5. Model synchronization.

plication framework like EMF is that some features come for free: Persistence of model states (i.e. saving and loading). Furthermore, models generated by EMF implements the Observer pattern, meaning that changes made in the model by e.g. a result from ENFORCe will automatically be propagated to its observers (yielding the backwards synchronization from ENFORCe to the views). EMF also makes it easy to extend ROOTS' underlying EMF model to work with graph conditions.

ROOTS and AGG are software for typed attributed graphs while ENFORCe works on directed labeled graphs. For a smooth port, we disabled ROOTS' model editor and used a static model consisting of a singular symbol type and link type, each with one attribute: a string for labels.

3.1 Model Extensions

For graph conditions, we extended ROOTS' EMF model by a few interfaces, the inheritance hierarchy for those are shown in Fig. 6. The two interfaces `ROOTS` and `Rule` are part of the original ROOTS model (see [14] for details), the others are our extensions. `ROOTS` collects graphs, rules, etc, into a graph transformation system, and `GCROOTS` adds nested graph conditions and rules with left and right application conditions to these transformation systems.

The already existing interface for negative application conditions were not expanded upon, since its view in ROOTS is too different from the morphism

view we are aiming for – our morphism view is more similar to ROOTS' rules. `GCMorphism` is a generalization of the nodes with morphisms in the tree view and therefore extends `Rule`, this allows us to reuse the components for editing a rule's left and right hand side to the editing of the morphism's domain and codomain. `NestedGraphCondition` specifies the nesting of graph conditions and is therefore a supertype of all types that are represented as nodes in the tree view. The original ROOTS model already supports binary Boolean formulas, but our conditions are best represented by junctions on sets of conditions. Boolean formulas are therefore represented by the new types `GCConjunction`, `GCDisjunction`, and `GCNegation`.

`GCRule` extends normal rules with left and right application conditions, `GCLAC` and `GCRAC` – both consisting of a set of `NestedGraphCondition` objects. The interface `GCContainer` specifies the possible contexts of where a `NestedGraphCondition` may occur.

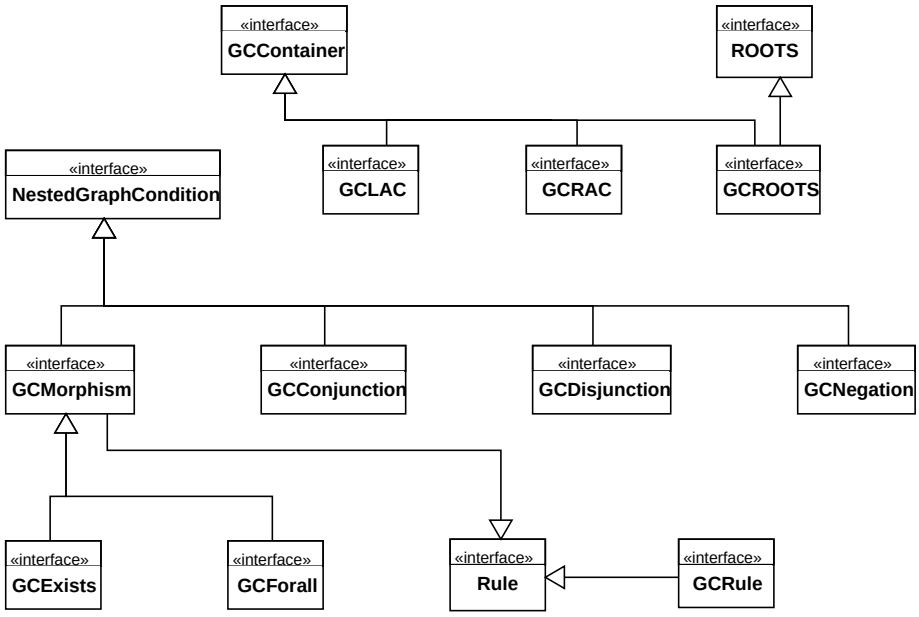

Fig. 6. Model extension for editing graph conditions.

3.2 View Extensions

Recalling the two views discussed in Sect. 2, we now discuss a few issues regarding their implementation in ROOTS. The tree view was implemented as an extension to ROOTS' existing tree editor for graphs, rules, etc. As an example, we show

the tree view from Fig. 4 (a) in the bottom of the left area in Fig. 7, named GraphCondition there. The right area of Fig. 7 shows the morphism view – the reused rule view – of the morphism identified by b_2 in Fig. 4 (b).

Since graph conditions can also be used in a rule context, the corresponding visual additions where made to a special kind of rule with left and right application condition. Figure 7 compares the previous rules with negative application conditions (named Rule in that figure) and the new rules with left and right application conditions (there named Rule w. a.c.).

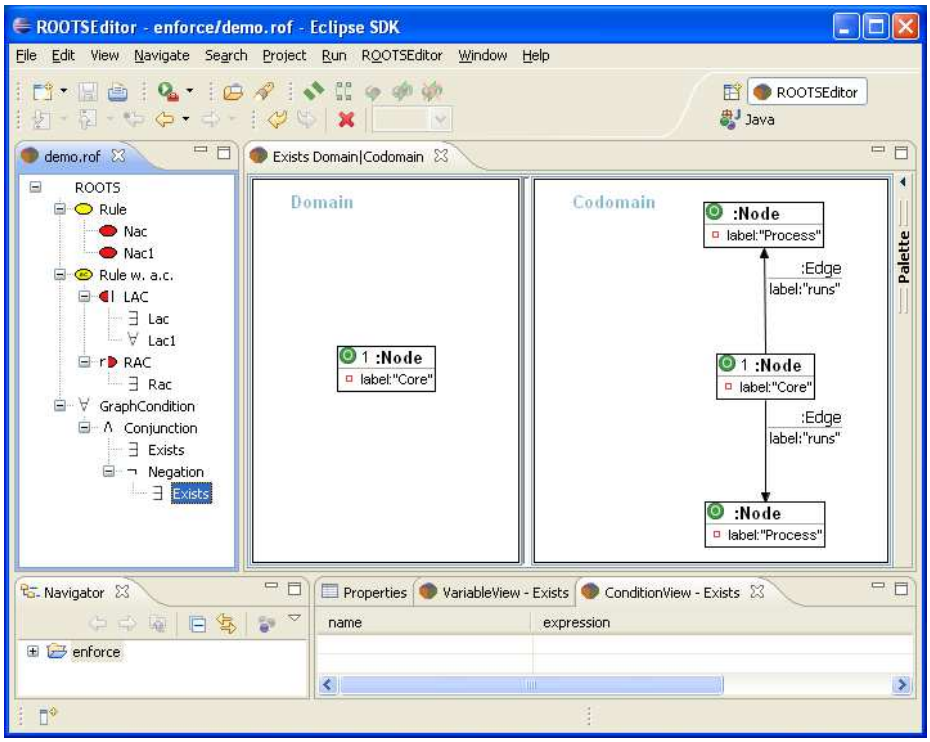

Fig. 7. Editing a nested constraint.

The morphism view is implemented by making some extensions to the rule editor of ROOTS, e.g. when a nested condition is created, it is by default set to the identity morphism of the codomain of its closest ancestor of a `GCMorphism` subtype. Commands for editing the trees and opening the morphism editor from the tree view are integrated with ROOTS' standard mouse operations.

4 Summary

We reported on an integration of ENFORCe and the ROOTS GUI for graph transformation systems and described an extension for graph conditions. Our layout approach were based on the nesting structure of graph conditions and was separated into two views, one for nesting and one for morphisms. We could therefore reuse existing ROOTS components and visually integrate the extension with other GUI elements. The resulting editor is however still quite similar to the representation used in e.g. [7].

It would be interesting to have a plug-in architecture in ROOTS that allows new features to be integrated more easily. Plug-ins would initially report to ROOTS on the particular category they support. ROOTS could then better layout objects of the particular category and report on what tool features are available to which categories. This would also allow the editor to be generalized to work on nested high-level conditions. Since the tree structure and mapping mechanism can work on objects from any HLR category, only the object visualization needs to be updated for new categories. Editor generators like GMF [13], Tiger [15], and DiaMeta [16] could provide interesting additions to ROOTS.

It should be investigated how nested graph conditions are best exported into an exchange format. GTXL [17], an exchange format for graph transformation systems, allows for conditions with nesting to occur in different contexts (i.e. as constraints and application conditions) and could be used as a basis. This would allow for experimentally comparing nested graph conditions to implementations of similar concepts like e.g. [18].

References

1. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001). Volume 2030 of Lecture Notes in Computer Science., Springer-Verlag (2001) 230–245
2. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific (1997)
3. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
4. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 3: Concurrency, Parallelism and Distribution. World Scientific (1999)
5. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science (2008) To appear.
6. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae **74(1)** (2006) 135–166
7. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: Graph Transformations (ICGT 2006). Volume 4178 of Lecture Notes in Computer Science., Springer-Verlag (2006) 445–460

8. Azab, K., Habel, A., Pennemann, K.H., Zuckschwerdt, C.: ENFORCe: A system for ensuring formal correctness of high-level programs. In: Proc. of the Third International Workshop on Graph Based Tools (GraBaTs'06). Volume 1 of Electronic Communications of the EASST. (2006)
9. Jurack, S., Taentzer, G.: ROOTS: An Eclipse Plug-in for Graph Transformation Systems based on AGG. In: Pre-proc. of the Third International Symposium, Applications of Graph Transformations with Industrial Relevance (Agtive 2007). (2007)
10. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: Language and environment. In: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2. World Scientific (1999) 551–603
11. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool integration at the meta-model level within the fujaba tool suite. International Journal on Software Tools for Technology Transfer (STTT) **6** (August 2004) 203–218
12. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Graph-Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science., Springer-Verlag (1979) 1–69
13. Foundation, T.E.: Eclipse application frameworks. Web pages available at `http://www.eclipse.org/{emf,gef,gmf}` Visited June 3, 2008.
14. Jurack, S.: Konzeption und Implementirung einer Entwicklungsumgebung für regelbasierte Graphtransformationssysteme basierend auf AGG und Eclipse (2007) Master thesis, TU Berlin.
15. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Towards graph transformation based generation of visual editors using eclipse. Electr. Notes Theor. Comput. Sci **127**(4) (2005) 127–143
16. Minas, M.: Generating meta-model-based freehand editors. In: Proc. of the Third International Workshop on Graph Based Tools (GraBaTs'06). Volume 1 of Electronic Communications of the EASST. (2006)
17. Lambers, L.: A new version of GTXL: An exchange format for graph transformation systems. In: Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004). Volume 127 of Electronic Notes in Theoretical Computer Science. (2005) 51–63
18. Rensink, A.: Representing first-order logic using graphs. In: Graph Transformations (ICGT'04). Volume 3256 of Lecture Notes in Computer Science., Springer-Verlag (2004) 319–335