Solving the Hello World Case with GROOVE

Amir Hossein Ghamarian Maarten de Mol Arend Rensink Eduardo Zambon

Department of Computer Science
University of Twente, The Netherlands
`rensink@cs.utwente.nl`

1 Introduction

This report presents a solution to the Hello World case study using GROOVE. We provide and explain the grammar that we used to solve the case study. Every requested question of the case study was solved by a single rule application.

2 GROOVE

GROOVE[1] [1] is a general purpose graph transformation tool set that uses simple labelled graphs. The core functionality of GROOVE is to recursively apply all rules from a predefined set (the graph production system – GPS) to a given start graph, and to all graphs generated by such applications. This results in a *state space* consisting of the generated graphs.

The main component of the GROOVE tool set is the Simulator, a graphical tool that integrates (among others) the functionalities of rule and host graph editing, and of interactive or automatic state space exploration.

2.1 Host Graphs

In GROOVE, the host graphs, i.e., the graphs to be transformed, are simple graphs with nodes and directed labelled edges. In simple graphs, edges do not have an identity, and therefore parallel edges (i.e., edges with same label, and source and target nodes) are not allowed. Also, for the same reason, edges may not have attributes.

In the graphical representation, nodes are depicted as rectangles and edges as binary arrows between two nodes. Node labels can be either node types or flags. Node types [resp. flags] are displayed in **bold** [resp. *italic*] inside a node rectangle.

2.2 Rules

The transformation rules in GROOVE are represented by a single graph and colours and shapes are used to distinguish different elements. Figure 1 shows a small example rule.

- **Readers.** The black (continuous thin) nodes and edges must be present in the host graph for the rule to be applicable and are preserved by the rule application;
- **Embargoes.** The red (dashed fat) nodes and edges must be absent in the host graph for the rule to be applicable;

[1]Available at `http://groove.cs.utwente.nl`

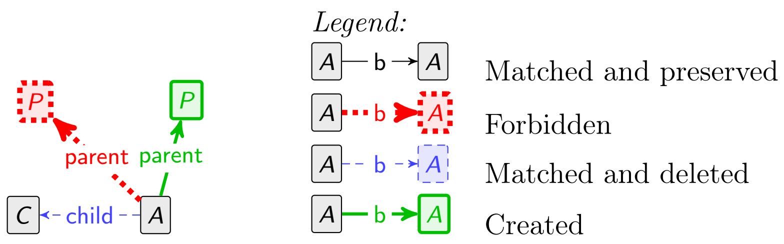


Figure 1: Example GROOVE rule and legend

- **Erasers.** The blue (dashed thin) nodes and edges must be present in the host graph for the rule to be applicable and are deleted by the rule application;
- **Creators.** The green (continuous fat) nodes and edges are created by the rule application.

Embargo elements are usually called Negative Application Conditions (NACs). When a node type or flag is used in a non-reader element but the node itself is not modified, the node type or flag is prefixed with a character to indicate its role. The characters used are $+$, $-$, and !, for creator, eraser, and embargo elements, respectively.

Additional notation and functionalities of the tool are presented along with the developed solution for the case.

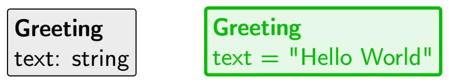
3 Solution

In this section we describe our solution to the case study.

3.1 Hello world!

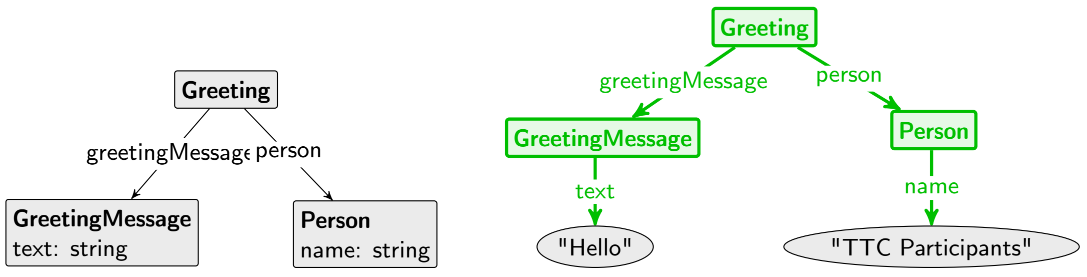
Metamodels are modelled as type graphs in GROOVE. Each node in a type graph corresponds to a node type; some have associated attributes. Types shown in ***bold italic*** inside dashed nodes are abstract. Edges with open triangular arrows indicate type inheritance.

- The greeting metamodel of the case study is modelled by the type graph in Figure 2(a). The rule which makes a node of type greeting is shown in Figure 2(b). The "text" element of the greeting type is defined as a string attribute.
- The type graph presented in Figure 3(a) represent the metamodel given in Fig. 2 of the case study. The "text" element of type GreetingMessage and the "name" element of type Person are modelled using string attributes. Figure 3(b) models the rule that generates the required graph which complies with the type graph.
- Every rule in GROOVE can have a print format property which describes the format in which the rule writes to the standard output when it is applied. The value of attributes with parameters can be written to the standard output by a format similar to that of printf. In this case the text attribute of GreetingMessage and the name attribute of Person have parameters 0 and 1 respectively. The print format property of the helloMessage rule is "The output is %s %s %n", in which each "%s" refers to the value of a parameter based on its order of appearance in the print format. The application of this rule prints "The output is Hello TTC Participants" to the standard output.



(a) Greeting type graph (b) Make greeting node rule

Figure 2: Greeting type graph and make greeting node rule



(a) Greeting message type graph (b) Make greeting message rule

Figure 3: Greeting message type graph and rule

3.2 Counting questions

All questions in this section are to compute the number of occurrences of different subgraphs in the context graph. GROOVE allows the use of nested universal and existential quantification in the rule description. All occurrences of subgraph can be captured by a universal quantifier. Each universal quantifier can report the number of found matches in an "int" attribute which is connected to the quantifier by a special edge with the label "count". Similar to the previous section, by adding a parameter to the attribute, we can print the number of matches to the standard output.

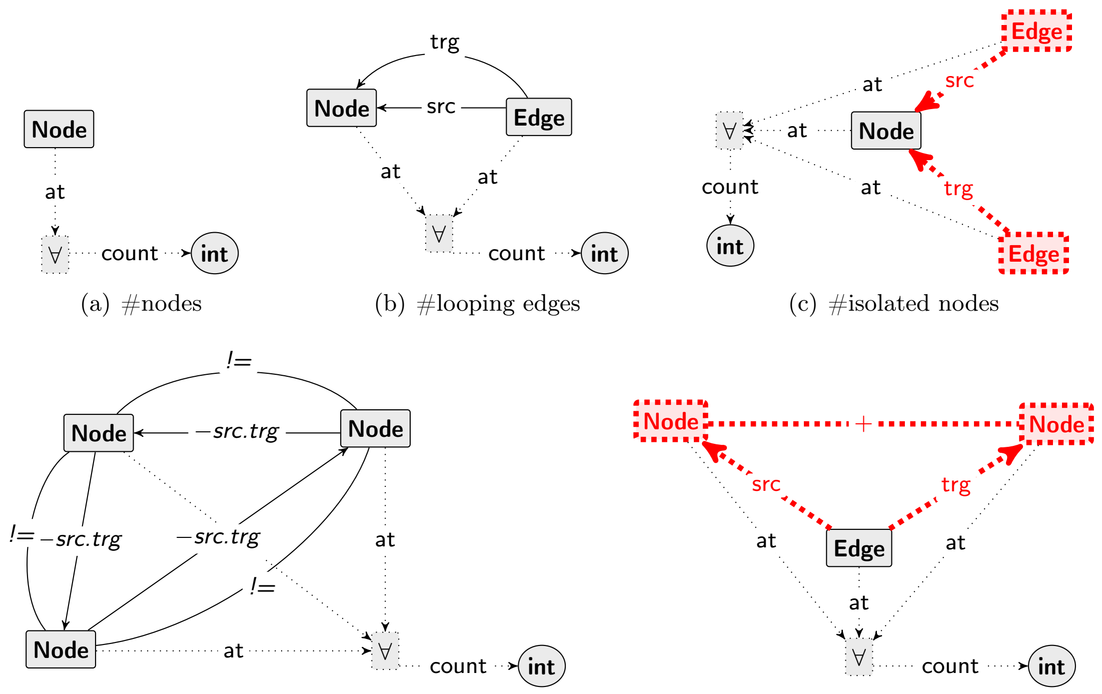
#nodes As represented in Figure 4(a), the number of nodes of type Node can be found by a universally quantified node of type Node.

#looping edges Similarly, for finding the loop edges, we only need to universally quantify a looping edge (see Figure 4(b)).

#isolated nodes An isolated node is described by a node of type Node with two NACs for both *src* and *trg* (shown in Figure 4(c)). A universal quantifier finds and counts all occurrences.

#cycles of three Figure 4(d) shows the rule for counting the number of cycles with three nodes. A cycle with three nodes is trivially described by three connected nodes. The edge with the label "!=" (injectivity constraint) ensures that the nodes are pairwise different, and the edges with label "-*src.trg* " are abbreviations for paths with two edges, with labels *src* and *trg*, respectively. The first edge with label src must have a reverse direction. The number of occurrences can be counted by universally quantifying this cycle.

#dangling edges This rule is very similar to the isolated nodes case. Dangling edges are nodes with type Edge of which **at least one** of its outgoing edges (*src* or *trg*) is missing, however, isolated nodes are nodes of type Node with **both** incoming *src* and *trg* edges missing. Therefore, for specifying dangling edges we need to specify a disjunctive relation between the two NACs for the absence of *src* and *trg*. This disjunctive relation between NACs is specified by an edge with label "+" in Figure 4(e).



(a) #nodes (b) #looping edges (c) #isolated nodes

(d) #cycles of three (e) #dangling edges

Figure 4: Counting rules

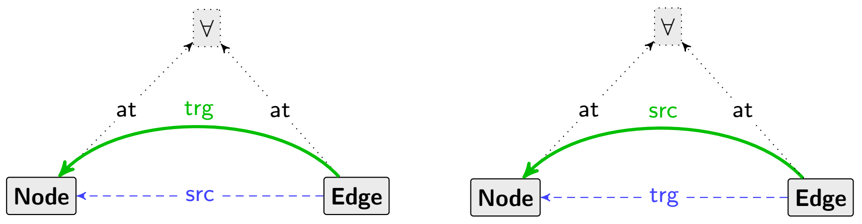


Figure 5: Reverse edges rule

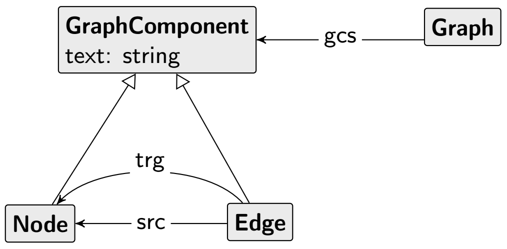
3.3 Reverse edges

This rule has two parts, the first part shown on the left-hand side of Figure 5 replaces all edges with label *src* with edges with label *trg*. The second part, shown in the right-hand side of Figure 5 replaces all edges with label *trg* by edges with label *src*. Note that this rule also complies with dangling edges as the rule does not require an edge to have both *src* and *trg*. Moreover, because of the universal quantifier the rule is applied at once, and all edges are reversed by one rule application. Performing this task without the use of a universal quantifier needs extra control mechanism to avoid applying the rule forever.

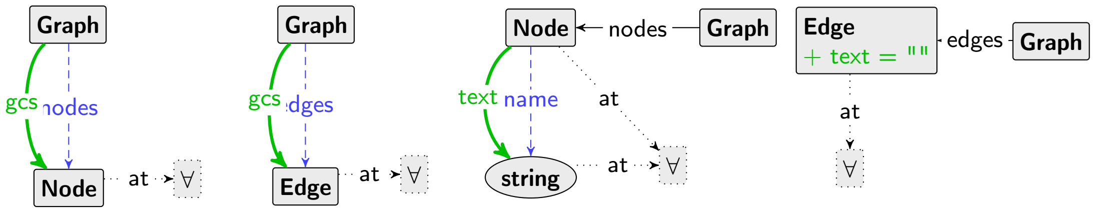
3.4 Simple migrations

GROOVE allows multiple type graphs to be used. In the migration case we enable both the source and the target type graphs.

Graph component migration The target type graph in GROOVE shown in Figure 6(a) is notationally very similar to the one given in the case study description. The migration rule shown in Figure 6(b) consists of four parts, the first two parts in the left rename

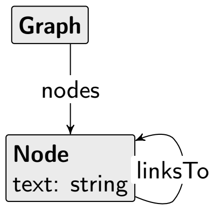


(a) Type graph with graph components

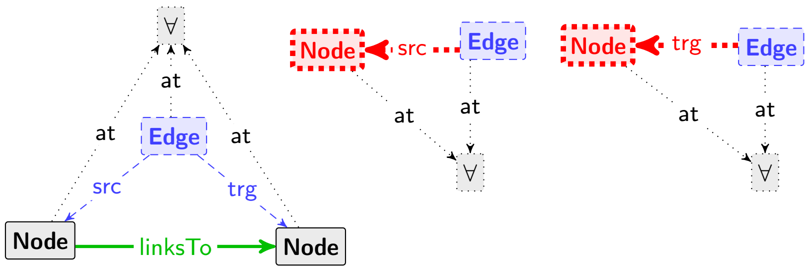


(b) Migration rule

Figure 6: Type graph and the migration rule



(a) Type graph with Edge nodes removed



(b) Migration rule

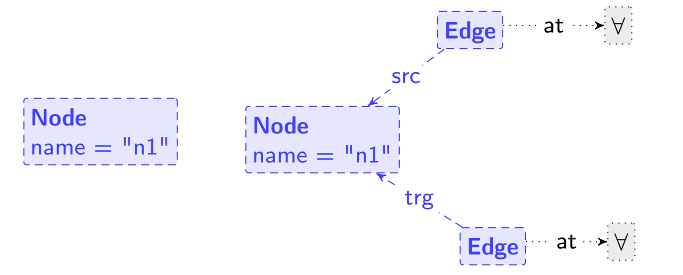
Figure 7: Type graph and the migration rule

the labels *nodes* and *edges* to *gcs*, the third part renames the "name" attribute of nodes of type Node to "text", and finally the fourth part of the rule adds an attribute "text" to nodes of type Edge and initializes it with an empty string. All parts of the rule are universally quantified.

Topology changing migration The type graph for the topology-changing migration case is shown in Figure 7(a). The migration rule is depicted in Figure 7(b). This rule has also three parts, the first part adds a new edge with the label *linksTo* between any two nodes of type Node and removes the node Edge. The other two parts of the rule are to remove the dangling edges. Similar to the previous migration, all parts of the rule are universally quantified.

3.5 Delete node with specific name

Node deletion Deleting a node with a specific name can be easily done in GROOVE. In this case, we only need to have an eraser node with attribute name "n1". Connected edges with labels *src* and *trg* are automatically deleted as GROOVE uses single push-out rewriting formalism. (show in Figure 8(a).)



(a) Delete node with name "n1"
(b) Delete node with name "n1" and its incident edges

Figure 8: Type graph and the migration rule

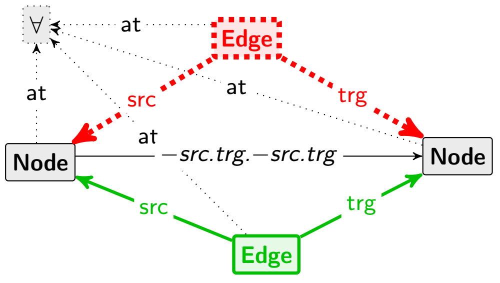


Figure 9: Transitive closure rule

Node and incident edges deletion In this case, nodes of type Edge which are connected to the node with name "n1" must be explicitly deleted. Such nodes can be deleted using two separate universal quantifiers, one for the edge nodes connected with an edge *src* and one for edge nodes connected with an edge *trg* (see Figure 8(b)).

3.6 Insert transitive edges

The rule for inserting transitive edges is shown in Figure 9. The rule checks for the existence of a path with length two (two edge nodes) and the lack of a path of length one (one edge node). The path with length two is specified using the regular expression "-*src.trg*.-*src.trg* ", as the identity of the intermediate edge nodes is not important. Please note that this is just an abbreviation, we can instead specify a path by explicitly specifying the edge nodes. The lack of existence of a path of length one is specified by a NAC. The insertion of a new edge is shown by a creator edge node and two edges. Again all elements must be universally quantified as we want to insert transitive edges for the whole graph.

4 Conclusion

In this report we presented a GROOVE solution to the Hello World case. We showed that all requested operations including the optional ones can be solved easily. Each task is solved using only one rule application of a single rule. All rules look very simple and contain few nodes only. No control language or any other control mechanism was used and all solutions use solely graph transformation mechanisms of GROOVE.

References

[1] Ghamarian, A., de Mol, M., Rensink, A., Zambon, E., and Zimakova, M.; *Modelling and analysis using* GROOVE . International Journal on Software Tools for Technology Transfer (STTT). Springer – Berlin, March 2011, http://dx.doi.org/10.1007/s10009-011-0186-x.