User Manual for the GROOVE Tool Set

Arend Rensink, Iovka Boneva, Harmen Kastenberg and Tom Staijen

Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE Enschede, The Netherlands
{rensink,bonevai,h.kastenberg,staijen}@cs.utwente.nl

June 11, 2009

Contents

1 Introduction

GROOVE is a project centered around the use of simple graphs for modelling the design-time, compile-time, and run-time structure of object-oriented systems, and graph transformations as a basis for model transformation and operational semantics. This entails a formal foundation for model transformation and dynamic semantics, and the ability to verify model transformation and dynamic semantics through an (automatic) analysis of the resulting graph transformation systems, for instance using model checking.

This manual constsis of some download and installation instructions and a manual for using the tools included in the GROOVE tool set. The latter also explains the format used for graphs and graph transformations. Together with some examples, this should allow you to get started with GROOVE.

1.1 Toolkit Components

The GROOVE tool set includes an editor for creating graph production rules, a simulator for visually computing the graph transformations induced by a set of graph production rules, a generator for automatically exploring state spaces, and an imaging tool for converting graphs to images.

1.2 Getting it running

Since the entire GROOVE tool is written in Java, getting it running is extremely easy.

1.2.1 Download

The GROOVE tool is distributed under the Apache License, Version 2.0. A copy of this license is available on `http://www.apache.org/licenses/LICENSE-2.0`. The latest GROOVE build can be downloaded from the GROOVE sourceforge page:

`http://sourceforge.net/projects/groove`

There are some different distributions of the GROOVE tool set available on the sourceforge site. The *groove-bin+lib* package includes all the libraries below. The *groove-bin* package is identical but without the libraries. The *groove-src* package only includes the sources of the groove project. There are also some examples available in the *groove-samples* package. The *groove-doc* package consists of some publications about GROOVE and the theory behind it.

The GROOVE library depends on some other libraties, namely:

- Castor - `http://www.castor.org/` (Version included: 0.9.5.2)
- JGraph - `http://www.jgraph.com/` (Version included: 5.9.2)
- JGraphAddons (Version included: 1.0.4)
- EPS Graphics Library - `http://sourceforge.net/projects/epsgraphics/` (Version included: 1.0.0)
- Xerces - `http://xerces.apache.org/xerces2-j/` (Version included 2.6.0 (Impl))

1.2.2 Installation on Windows systems

To use the GROOVE tool set under windows, download the bin+lib package from the download site explained above and unzip it to any location on your computer. We will refer to the destination folder as GROOVE_PATH. Under GROOVE_PATH, you will find a directory *bin* which contains some batchfiles, that refer to the tools contained by the GROOVE tool set (Editor.bat, Simulator.bat, Generator.bat and Imager.bat). Inside the batchfiles, a variable GROOVE_BIN_DIR is set. Make sure the value it is set to is your local GROOVE_PATH.

Then make sure you have a JRE (Java Runtime Environment) version 5.0 (or higher) installed and that you have it in your PATH. You can download a JRE from `http://java.sun.com`.

Now double-click on a batch-file to run the desired tool.

1.2.3 Installation on UNIX based systems

To use the GROOVE tool set under windows, download the bin+lib package from the download site explained above and unzip it to any location on your computer. We will refer to the destination folder as GROOVE_PATH. Under GROOVE_PATH, you will find a directory *bin* which contains some executable shell-scripts, that refer to the tools contained by the GROOVE tool set (Editor, Simulator, Generator and Imager). Inside the shell-scripts, a variable GROOVE_BIN_DIR is set. Make sure the value it is set to is your local GROOVE_PATH.

Then make sure you have a JRE (Java Runtime Environment) version 5.0 (or higher) installed and that you have it in your PATH. You can download a JRE from `http://java.sun.com`.

Now execute a shell-script to run the desired tool.

2 Basic Concepts

When working in GROOVE, in particular when creating or modifying rule systems, it is important to reaslise that there are two different display modes for graphs and rules: the one that you usually see in the Simulator, and the one of the Editor. We call these the Display and Edit views, respectively. Historically, the Edit view only exists because we never took the time to create a better graph editor; instead, we ripped off a prototype editor provided for free with the JGRAPH library.

2.1 Graphs

GROOVE uses edge-labelled, directed graphs. Graph nodes are depicted as boxes, and edges as arrows between them. Self-edges (often also called loops, i.e., edges whose source and target nodes coincide) are often not displayed as arrows; instead, node labels are used to represent labels of self-edges. It is important

to realise that these are formally *exactly the same*. For instance, the left and right hand side of Figure 1 depict exactly the same graph.

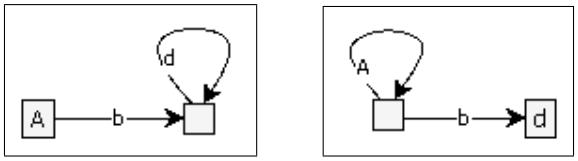


Figure 1: Node labels represent self-edges, hence these two views depict the same graph.

In the GROOVE views, both nodes and edges can have multiple labels. Multiple node labels are given through a vertical list; multiple edge labels are given in the form of a comma-separated list. In both cases, these lists actually represent multiple edges; for instance, the left and right hand sides of Figure 2 depict the same graph.

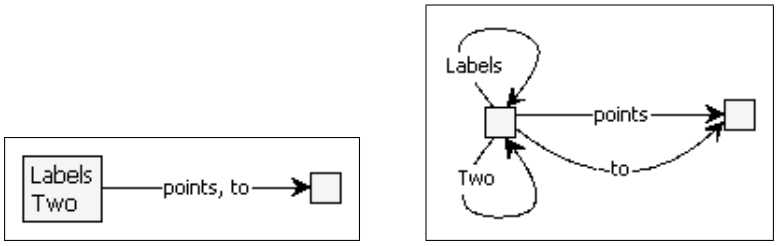


Figure 2: Multiple labels represent multiple edges, hence these two views depict the same graph.

Labels in graphs. There are two restrictions on the character sequences that can be used as labels in graphs:

- Colons (":") may not be used arbitrarily. This is because colons are used to separate *label prefixes* (see below) from the remainder of the label. This restriction can be circumvented by *starting* the label with a colon: this initial colon is then not counted to be part of the label proper, and the remainder is not parsed for colons. (Thus, an initial colon serves as an "escape" character precisely as an initial single quote serves as an escape in Excel.)
- Whitespace other than simple spaces, such as tabs and newlines, cannot be included in labels.

2.2 Rules

Formally, rules consist of left hand sides, right hand sides and negative application conditions (NACs), all of which are different graphs, connected by morphisms. In GROOVE these graphs are combined into one single graph, and colour coding is used to distinguish the original components. As a consequence, a GROOVE view of a rule has the following kinds of elements:

Readers. These are nodes and edges that are in both the LHS and the RHS. In both the editor and the display view, they are depicted just like ordinary graph elements; hence, the outlines are thin and black and the font colour is black.

Erasers. These are nodes and edges that occur in the LHS but not the RHS, meaning that they must be matched in order for the rule to apply, but by applying the rule they will be deleted. In the Display view, such elements are depicted by a thin, dashed blue outline and blue text. In the Edit view, erasers are distinguished by a special prefix "del:". For eraser nodes, this prefix should appear *on its own* as a node label; for eraser edges, the prefix is followed by the edge label.

Creators. These are nodes and edges that occur in the RHS but not the LHS, meaning that they will be created when the rule is applied. In the Display view, such elements are depicted by a slightly wider, solid green outline (light grey in a black-and-white representation) and green text. In the Edit view, creators are distinguished by a special prefix "new:". For creator nodes, this prefix should appear *on its own* as a node label; for creator edges, the prefix is followed by the edge label.

Embargoes. These are nodes and edges that are in a NAC, but not in the LHS. This means that they *forbidden*: their presence in the host graph will prevent the rule from being applied. In the Display view, such elements are depicted by a wide, dashed red outline (darker grey in a black-and-white representation) and red text. In the Edit view, creators are distinguished by a special prefix "not:". For embargo nodes, this prefix should appear *on its own* as a node label; for embargo edges, the prefix is followed by the edge label.

Another thing to note is that if a node plays any of the roles of eraser, creator and embargo, its incident edges implicitly also have this role. Thus, in that case the corresponding prefix can be omitted in the Edit view.

For example, Figure 3 shows the Edit and Display views of a rule which contains all of these types of elements.

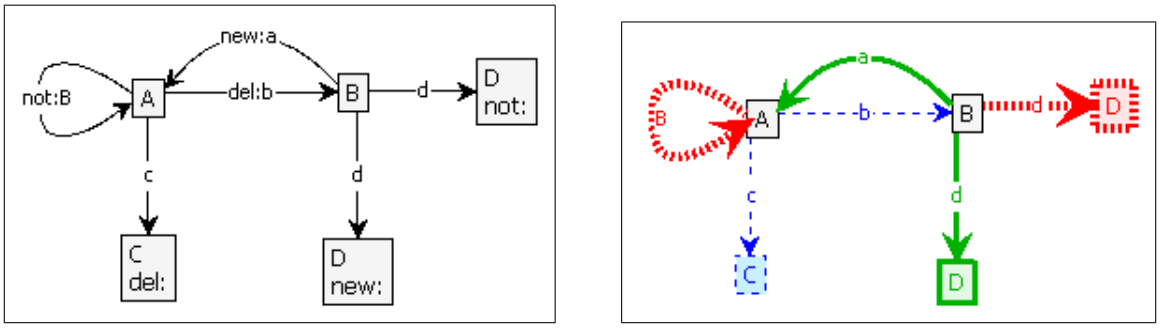


Figure 3: Edit and Display views of a simple rule.

Labels in rules. Label parsing in rules is more complicated than in graphs, because there are many more special labels (see below for a discussion). The following points should be noted.

- The rule for the use of colons is the same as for graphs: when an (unquoted) colon is used as part of a label, there should be a single initial colon preceding the entire label; this initial colon is not considered to be part of the label itself.
- In addition to the above, whenever the spacial characters ' (single quote), \ (backslash), ? (question mark), ! (exclamation mark), = (equality sign), or { and } (opening and closing curly braces) are used literally within labels, i.e., not in their role as special characters, the whole label must be single-quoted. The surrounding single quotes are themselves not considered to be part of the label.

- The backslash ("\") serves as an escape character within a single-quoted label: any next character (including the backslash itself) is interpreted literally rather than as a special character. This is especially needed to use single quotes within single-quoted labels.

For instance, the label '\\?\" (ending on two single quotes) in a rule matches the label \?' in a graph.

Rule names. Rules have names. The names are essentially identifiers. The actual constraints on rule names are quite flexible: any string that can be used as a file name but does not contain spaces or periods is allowed as a rule name. However, it is *recommended* to stick to rule names that are valid identifiers:

- Start a rule name with a letter — by convention a small letter;
- Restrict the remaining characters to letters, digits, underscores or dollar characters.

Rule names can impose a hierarchical structure, similar to the package structure of qualified Java class names. For instance, the name "a.b" stands for rule "b" in package "a". This mechanism is only there for the purpose of structuring larger sets of rules; the structure does not change the meaning of the rule system (see also Section **??** below).

The GROOVE samples contain the following rule systems showing various aspects of the features discused above:

- circular-buffer, a simple data structure with two rules, containing creators, erasers and embargoes;
- loose-nodes, showing that node labels are just self-edges which can be added to existing, non-labelled nodes.

2.3 Negations

Another way to forbid an edge is by inserting an exclamation mark in front of its label. This therefore has the same effect as the "not:" prefix, but it can only be used for edges. Moreover, negations can also be used *within* embargoes, achieving a double negation. For instance, Figure 4 expresses that the rule may only be applied if the Bus has not already started (!start-self-edge), and there is *no* Pupil that is *not* in the bus (!in-embargo edge) — in other words, if all the pupils are in the bus.

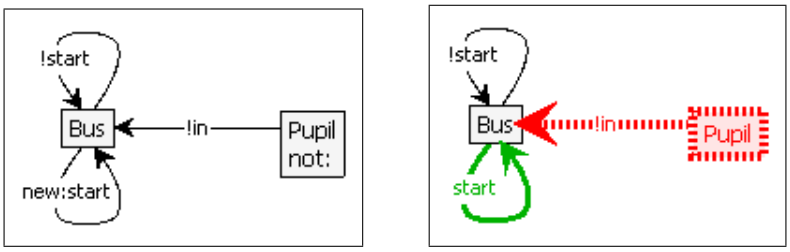


Figure 4: Edit and Display views of a rule with double negation.

Negations may only be used on reader and embargo edges; in fact, they would be meaningless when used on eraser or creator edges.

2.4 Equalities, Mergers and Injectivities

GROOVE has a special label "=" (the equals sign). When used between nodes in a rule, this expresses that the nodes are really the same, despite being depicted as different. Such equality labels may also be used on creator edges (which are then called *mergers*) and embargo edges (which are then called *injectitivies*). Moreover, they may be combined with negation.

Mergers. GROOVE rules can *merge* nodes. This is specified by a special edge labelled "new:=" between the nodes that are to be merged. The direction of the edge is irrelevant. When two nodes are merged, the resulting node receives the incident edges of both original nodes (including the self-edges). For instance, the rule in Figure 5 specifies that the start and final state of an automaton should be merged, while all incoming and outgoing transitions are preserved. See also rule system mergers in the GROOVE samples.

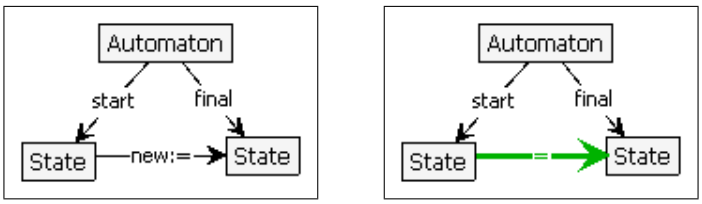


Figure 5: Edit and Display views of a rule with a merger.

Injectivities. In general, rules are not matched injectively — meaning that distinct LHS nodes may be matche dby the same host graph node. (See, however, Section 3.7 where we discuss how to set a global injectivity constraint through the system properties.) Local injectivity can be enforced by a special edge labelled "!=" or "not:=:" the end nodes of such an edge will always have distinct images. Note that the direction of the edge is irrelevant. For instance, the rule in Figure 6 specifies that a couple may only marry if they do not share parents.

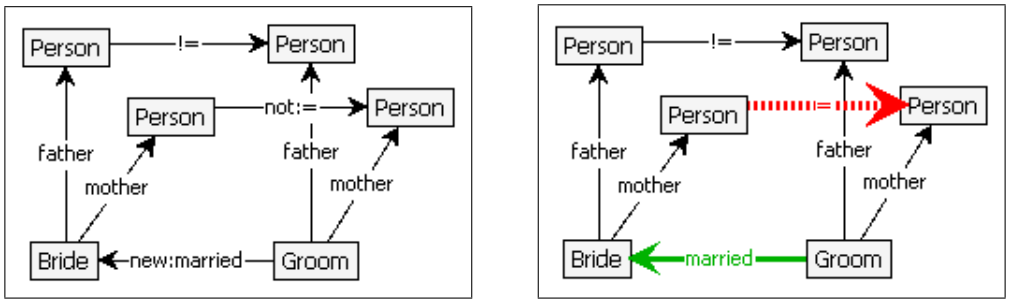


Figure 6: Edit and Display views of a rule with two injectivity constraints.

Counting. As with ordinary labels, the effect of negation (an explamation mark) in front of an equality is in principle the same as that of the embargo prefix — but again, negations can be used *within* embargoes. This has an important use in enabling *counting* in rules. For instance, Figure 7 specifies that a Plate may only be put in the Oven if it contains *exactly* three Rolls — no more and no less. The injectivity between the reader Rolls ensures that there are no less than two of them, whereas the embargo Roll with its injectivities ensures that there are no more than two. See also rule system counting in the GROOVE samples.

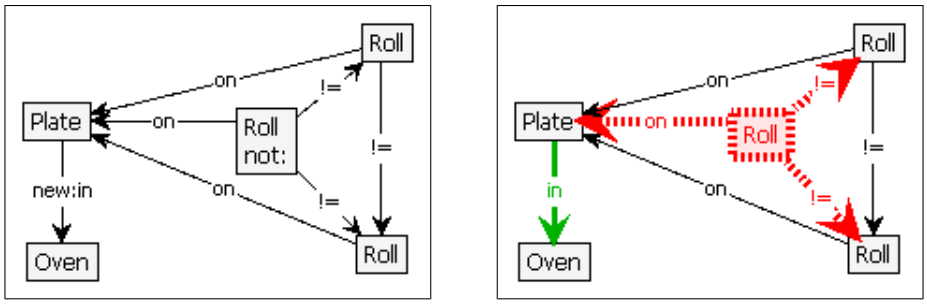


Figure 7: Edit and Display views of a rule with a counting constraint

2.5 Rule Comments

To document rules, GROOVE offers the possibility to add special nodes and edges that do not make a difference to the transformation. This is done through the prefix "rem:" (for "remark"), either on a node (as a stand-alone node label) or on an edge — just as for the prefixes we have seen so far. In the Display view, remark nodes and edges are orange, with a yellow background. For instance, Figure 8 is the same rule as Figure 7, augmented with remarks.

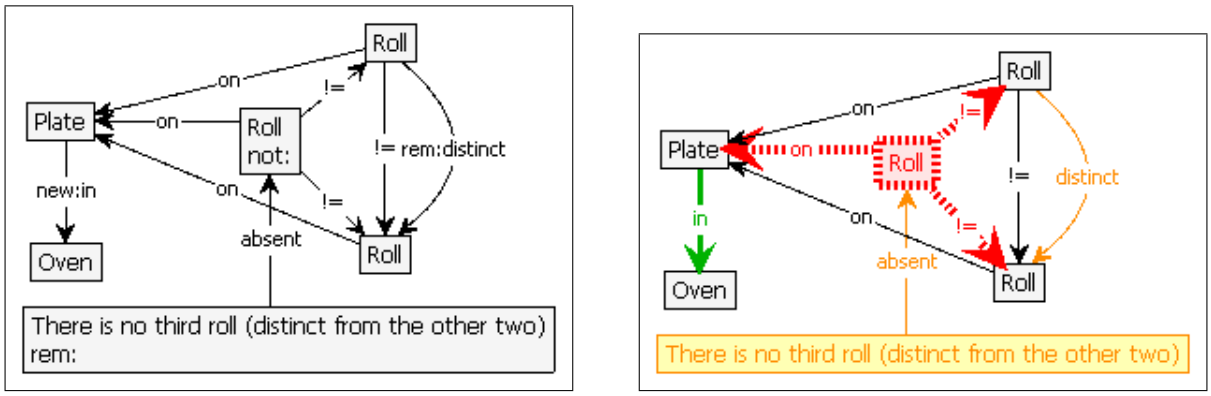


Figure 8: Edit and Display views of a rule with remark elements

2.6 Rule properties

Apart from the LHS, RHS and NAC, which are depicted graphically, a rule also has *rule properties*. These can be accessed and modified either from the Simulator or from the Editor. The most important of these properties is the *priority* of the rule.

Priorities. Rule priorities provide a (primitive) way to *schedule* the application of rules: as long as a high-priority rule is enabled, no lower-priority rules can be scheduled for application.

The default rule priority is 0. Creating rules with different priorities will change the rules overview in the Simulator: another, top level is introduced in this view, ordering groups of rules according to their priorities.

For instance, one can introduce a high-priority rule that just tests for the presence of an "Error"-labelled node, and does not modify the graph. Such a rule would automatically halt the transformation of a graph if some other rule introduces such an Error-node.

Rule system priorities in the GROOVE samples shows an example use of priorities.

Enabledness. A rule can be *disabled*, meaning that it is never scheduled for application. This can be very useful when developing a graph grammar, since it makes it easy to experiment with different versions of the same rule.

Comment. The rule comment provides another way to document a rule, in addition to the remark nodes and edges already described in Section 2.5.

2.7 Transition systems

During the evaluation of a set of rules, GROOVE "under water" builds up a so-called transition system, in which every graph plays the role of a state, and every rule application is interpreted as a transition. The transitions bear the names of the rules that have been applied as labels. The precise formatting of the transition labels can be controlled by two system properties, transitionBrackets and transitionParameters; see Section 3.7.

Rule systems and grammars A rule system is a set of rules, possibly with some addition information such as a control specification (see Section 3.5) and system properties (see Section 3.7). A *grammar* is a rule system together with a start graph. The default start graph, called start, is assumed to be available together with the rules; other start graphs can be specified or loaded in, depending on the circumstances.

The structure of rule names (consisting of substrings separated by periods, see Section 2.2) in fact imposes a hierarchy of name spaces on the rule system, but this hierarchy does not play a role in the evaluation of a graph grammar. In other words, the meaning of a graph grammar does not change if all the rules are arbitrarily renamed, including renamings that change the hierarchical structure.

3 Advanced Concepts

3.1 Wildcards and Variables

Wildcards are special edge labels that can be used in rules to stand for *arbitrary* labels. The basic wildcard is just a question mark "?": it is matched by any edge of which the source and target node also match. Wildcards can only be used in the LHS or NAC; in other words, a wildcard cannot be used on a creator edge — unless it is a *named* wildcard, see below.

Guarded wildcards. Wildcards can be *guarded* either by a list of allowed values or by a list of forbidden values:

- ?[a,b,c] stands for a wildcard that can only be matched by labels a, b or c (this is therefore the same as the regular expression "a|b|c"; however, in contrast to regular expressions, wildcards (when used on their own) may occur on eraser edges, and (when named) also on creator edges.
- ?[^a,b,c] stands for a wildcard that can be matched by any label *except* a, b or c.

Named wildcards. Finally, wildcards can have a *name*, in which case they act as *label variables*. The name directly follows the question mark, hence "?x" is a wildcard with name x. When such a wildcard is matched by a certain edge label, that label is considered to be the *value* of the variable for the duration of the rule application. The same label variable can occur multiple times within a single rule; the effect is that each of these occurrences must be matched by the same label.

Variable names can be freely chosen (except that they must adhere to the syntax rules of an identifier, i.e., start with a letter and consist only of letters, digits and underscores); they may in fact coincide with actual labels, though this must be considered bad practice. Variable names can also be combined with guards; for instance, "?x[^a,b,c]" is matched by any label except a, b or c; the matching label is then bound to x.

In contrast to ordinary wildcards, named wildcards can be used on creator edges, providing that a binding instance occurs in the LHS. This enables the *copying* of edge labels.

For instance, Figure 9 shows a rule which specifies that if the same label occurs as a self-edge on two different Persons, then this label should be added as a self-edge on a collector node labelled Duplicates, provided it is not already there. The label Person itself, however, is exempted from this treatment.

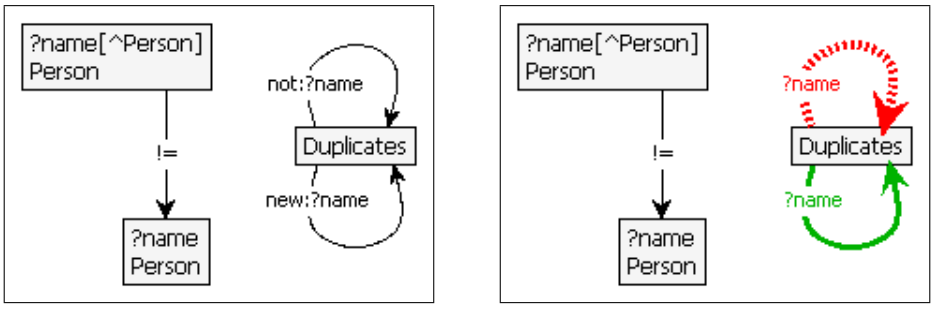


Figure 9: Edit and Display view of a rule with a named wildcard.

3.2 Regular Expressions

Rule edges can specify regular expressions over graph labels. Such a regular expression is matched by any chain of edges in the host graph of which the labels form a word recognised by the regular expression. Regular expressions may only be used on reader and embargo edges, never on erasers or creators.

Regular expressions are distinguished by surrounding curly braces. Thus, "{a.b}" specifies a regular expression (matched by two consecutive graph edges labelled "a" and "b") whereas "a.b" specifies a single edge with exactly that label. Regular expressions are built up from the following operators (for an overview see Table 10):

Atoms These are simple labels, to be matched precisely. Note that the syntax rules discussed in Section 2.2 must be followed whenever the label to be matched contains special characters.

Sequencing This is a binary infix operator, denoted ".", specifying that its left hand operator should match, followed by its right hand operator. Thus, a label sequence matches the regular expression $R_1.R_2$ if it can be split into two sequences, the first of which matches R_1 and the second R_2.

Choice This is a binary infix operator, denoted "|", specifying that one of its operands should match. Thus, a label sequence matches the regular expression $R_1|R_2$ if it matches either R_1 or R_2.

Expression	Meaning
label	Simple label; matched literally
=	Empty path/equality of nodes (see Section 2.4)
?	Wildcard, possibly named and/or guarded (see Section 3.1)
$R_1.R_2$	Sequential composition of R_1 and R_2
$R_1 \mid R_2$	Choice between R_1 and R_2
R^*	Zero or more repetitions of R
R+	One or more repetitions of R
-R	Inversion of R (matches R when followed backwards)
!R	Negation of R (absence of a match for R)

Table 10: Regular expressions

Star The star (or *Kleene* star) ("*") is a postfix operator that specifies that the preceding regular expression occurs zero or more times. Thus, a label sequence matches R^* if it can be split into zero or more subsequences, each of which matches R.

Plus The plus ("+") is a postfix operator that specifies that the preceding regular expression occurs one or more times. Thus, a label sequence matches R+ if it can be split into one or more subsequences, each of which matches R.

Inversion This is a prefix operator, denoted by the minus sign ("-"), specifying that its operand should be interpreted in reverse, *including the direction of the edges*. Thus, a sequence of edges matches -R if it matches R when followed backwards.

Equality An equality sign ("=") may be used as an atomic entity in a regular expression, in which case it stands for the empty word, or in other words, it is matched by an emtpy sequence of edges in the host graph. For instance, the regular expression "a|=" specifies that between two nodes there is an a-edge or the nodes coincide. Also, R^* has the same meaning as R+|= (for any regular expressions R).

Wildcard This is exactly as discussed in Section 3.1 above. Note that a named wildcard within a regular expression may in some circumstances fail to bind to any value: namely, when it is not necessarily matched, such as in a choice or star expression. If a variable is not bound, it may not be used on a creator edge.

Negation This is the same as discussed in Section 2.3. Negations are specified by a single exclamation mark ("!") preceding the entire regular expression. Thus, they cannot be used *inside* a regular expression. In fact, a negation is not properly part of the regular expression itself, since it is in itself not matched by anything; rather, it expresses the absence of a match for the actual regular expression.

For instance, Figure 11 shows a rule that specifies that a son should receive the name of one of his forefathers.

The GROOVE samples contain grammaers wildcards, variables and regexpr showing the use of the concepts discussed above.

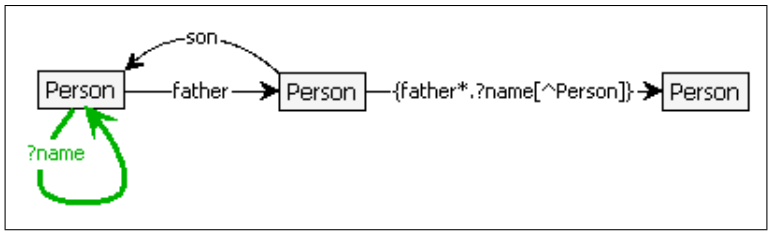


Figure 11: Rule with a regular expression.

3.3 Data Attributes

So far we have not discussed how to specify and manipulate data values, such as integers, booleans and strings. In GROOVE, as in other graph transformation tools, data is included in the form of *attributes*, which are essentially edges to special data nodes. The data nodes represent the actual data values.

Data values. Typically, graph nodes are abstractions of objects from the model space which somehow have an identity. That is, a graph can have multiple nodes that are indistinguishable when only their connecting edges are taken into account. This is not directly suitable for data nodes, however: for instance, every natural number exists only *once*, and it makes no sense to include multiple nodes all of which represent this single value. Thus, it is necessary to make a strict distinction between data nodes and ordinary graph nodes. In GROOVE, this is done in either of the following ways:

- If the concrete data value is known, then it is specified using a node label of the form "*type*:*const*", where *type* is the data type and *const* a denotation of its value. The available data types are int, bool, string and real. The denotation of the constants is the usual one; e.g., -1, 0, 1 etc. for int, true and false of bool and "text" for string.
- If the value is not known, for instance because the node occurs in the LHS and the value will only be established when matching the rule, then it should be labelled "attr:".

Data nodes can never be created or deleted and are always present (at least virtually); hence, they can only occur as readers.

Operations. In addition to specifying data values, we also need to manipulate them; that is, carry out calculations. This, too, is specified graphically, through the following special types of nodes and edges:

Product nodes, which essentially stand for *tuples* of data values. Product nodes are distinguished by the special label "prod:".

Argument edges, which lead from a product node to the data nodes that constitute the individual values of the tuple. Argument edges are labelled "arg:*num*", where *num* is the argument number, ranging from 0 to (but not including) the size of the tuple.

Operator edges, which lead from a product node to a data node representing the result of an operation performed on the elements of the tuple. Operator edges are labelled "*type*:*op*", where *type* is a data type (which are the same as for the data nodes) and *op* is an operation performed on the source node tuple; for instance, add (for a pair of int values), and (for a pair of bool values), or concat (for a pair of string values). Table 12 gives an overview of the available operations.

Type	Op	Meaning
bool	and	Conjunction of two boolean values
	or	Disjunction of two boolean values
	not	Negation of a boolean value
	eq	Comparison of two boolean values
	true	Boolean constant
	false	Boolean constant
int/real	add	Addition of two integer or real values
	sub	Subtraction of the second argument from the first
	mul	Multiplication of two integer or real values
	div	Integer (for int) or real (for real) division of the first argument by the second
	mod	Remainder after integer division (only for int)
	min	Minimum of two integer or real values
	max	Maximum of two integer or real values
	lt	Test if the first argument is smaller than the second
	le	Test if the first argument is smaller than or equal to the second
	gt	Test if the first argument is greater than the second
	ge	Test if the first argument is greater than or equal to the second
	eq	Comparison of two integer or real values
	neg	The negation of an integer or real value
	toString	Conversion of an integer or real value to a string
string	concat	Concatenation of two string values
	lt	Test if the first argument is lexicographically smaller than the second
	le	Test if the first argument is lexicographically smaller than or equal to the second
	gt	Test if the first argument is lexicographically greater than the second
	ge	Test if the first argument is lexicographically greater than or equal to the second
	eq	Comparison of two string values

Table 12: Data types and operations

In the Display view of rules, data nodes are depicted by ellipses and product nodes by diamonds. In the Display view of graphs, the attribute edges leading to data nodes as well as the data nodes themselves are not depcited as edges and nodes at all, but rather in the more familiar sttribute notation, as equations within the source nodes. (There is, however, an option in the Simulator to switch off the attribute notation and show data values as ellipsoid nodes; see Section 4.5.) For instance, Figure 13 shows the Edit and Display views of a rule that specifies the withdrawal from a bank account, provided the balance does not become negative.

Figure 14 shows the Edit and Display views of an attributed graph.

Algebras. Formally speaking, the operations listed in Table 12, as well as the data values discussed above, are actually operators and constant symbols out of a data *signature*. This signature is then interpreted by an *algebra*, which defines concrete values and functions for these symbols. There is a default or natural algebra for our signature, which is the one that we all know from mathematics; in a context where this is the only possible interpretation, the distinction between signature and algebra is actually irrelevant. However, GROOVE offers the possibility of slotting in another algebra instead: through the rule system properties (see Section 3.7) you can specify under which algebra the rules should be interpreted.

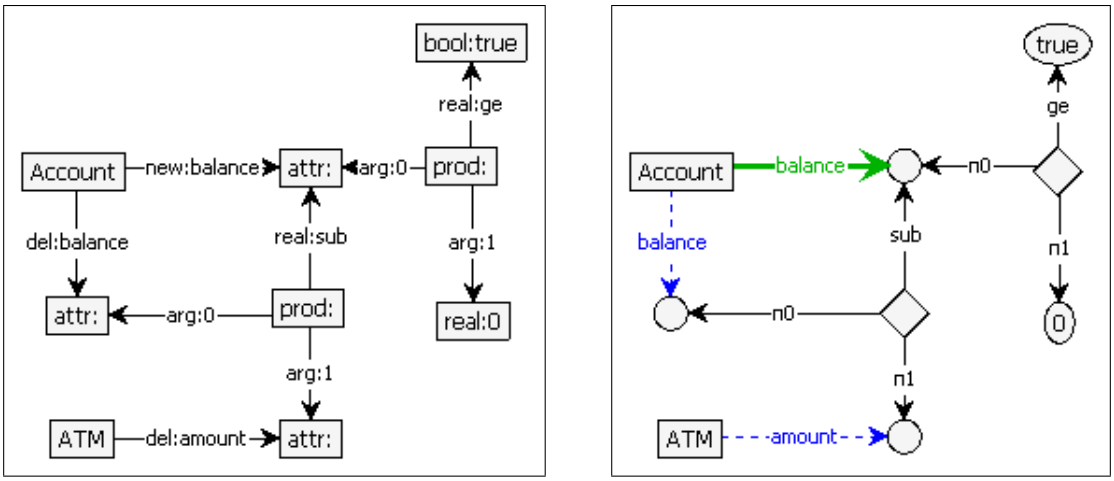


Figure 13: Edit and Display view of a rule using attributes.

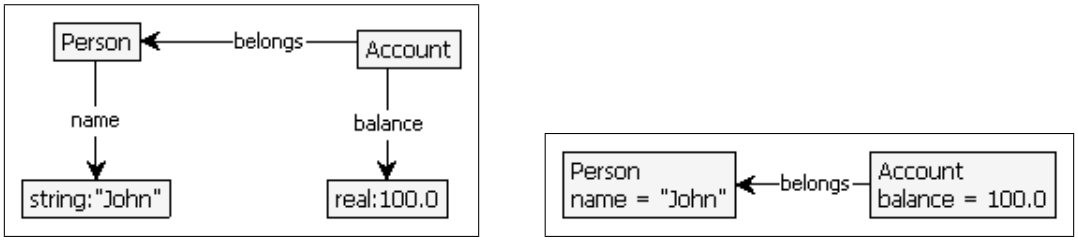


Figure 14: Edit and Display view of an attributed graph.

Currently, three choices are supported:

- The default algebra, which is actually implemented using the standard Java types int, double, boolean and String;
- The *point algebra*, in which each data type has exactly one value. Every constant and operation returns this value.
- The *big algebra*, in which integers have arbitrary precision and reals have 34 digits of mantissa (which is twice the precision of Java doubles).

In the default algebra, comparison of reals (using eq, geq etc.) has a *tolerance* of 10^{-15}. In other words, if the difference between two values is less than 10^{-15} times any of these values, then the values are considered to be equal. This is so as to avoid the phenomenon that rounding errors result in an artificial difference between values that would otherwise be equal. In the case of the big algebra, the tolerance is 10^{-30}.

An example of a grammar with attributes can be found in the attributed-graphs grammar in the GROOVE samples.

3.4 Rule Parameters

Rule parameters provide a way to make information about a match visible in the transition system. A rule parameter is a node of LHS (on the implicit existential level, see Section 3.6) that is marked with the special

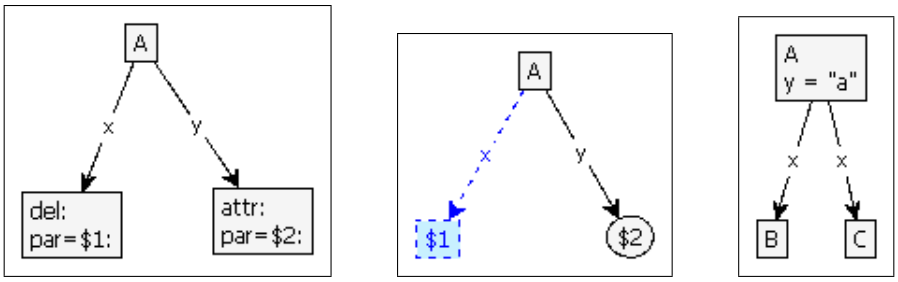


Figure 15: Edit and Display view of a parameterised rule, and a graph to which the rule is applicable.

prefix "par=$*num*", where *num* is a parameter number. Parameter numbers should form a consecutive sequence from 1 upwards; no parameter number may occur more than once in a given rule.

Node identities as arguments When a rule is eveluated, this results in a transition labelled by the rule name (see Section 2.7). However, if a rule has parameters, and if the transitionParameters property is set (see Section 3.7) then the transition labels are appended by lists of parameter values, being the node identities of the host graph nodes matching the parameter nodes.

Note that a node identity is normally not visible in a graph (unless it is switched on in the Simulator, see Section **??**). The node identities appearing as transition parameters are denoted "n*id*", where itseries id is the *node number* of the concrete graph node. *There is no guarantee that node identities are preserved among graphs!* This means that before and after a transition, the same node identity may refer to a completely different node. On the other hand, if a node identity appears on different parameterised transitions starting in the *same* graph, then it is certain that this refers each time to the same node of that graph.

Data values as arguments The situation is slightly different if the parameter node is an attribute node, for as discussed above, the identity of a data node is taken to be the data value itself. So, in that case, the data value is shown in the parameter list.

As an example, Figure 15 shows an Edit and Display view of a rule with two parameters, one a non-attribute eraser node and the other an attribute reader node. When this rule is applied to the graph also shown in the figure (where the node identity display has been switched on), there will be two transitions labelled parameters(n38152,"a") and parameters(n38153,"a"), respectively.

Anonymous parameters Declaring a node to be a rule parameter has another effect, besides putting the node identity on the transition label. Namely, those rule matches that map a parameter node to a different host graph node will *always* give rise to distinct rule applications, even if the rule effect is the same.

This is most noticeable in rules that do not modify the graph, i.e., in which the LHS and RHS coincide (no erasers and no creators). Such rules essentially encode *conditions* on the graph, i.e., they measure the existence of a match. Normally such an unmodifying rule is considered to have at most one application in any host graph, even if the LHS matches at different subgraphs of the host graph. However, if the rule has parameters, then matches that map the parameter nodes to distinct host graph nodes will give rise to distinct applications, with distrinct transition labels.

For the case that one needs distinct rule applications *without* having the node identity on the transition label, GROOVE offers the concept of an *anonymous parameter*. This is essentially a parameter without

number, in the editor specified by just the prefix "par:". An example is shown in Figure **??**: this rule, applied to the graph of Figure 15, will give rise to two distinct transitions, both labelled anonymousParameter("a"), which are self-loops on the state since neither rule application changes the graph.

Note that the display view does not show the anonymous parameter at all. We are still considering an appropriate visual representation.

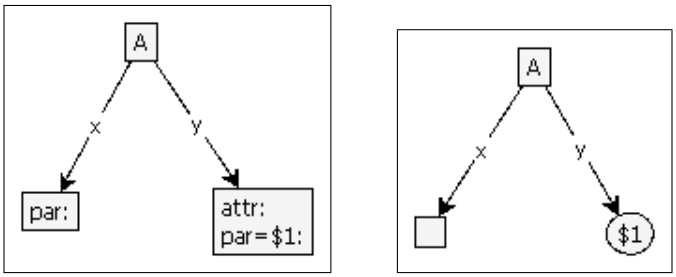


Figure 16: Edit and Display view of an unmodifying rule with an anonymous parameter.

Another example of a grammar with rule parameters can be found in the parameters grammar in the GROOVE samples.

3.5 Control

Control is about scheduling rule executions. It provides a much stronger mechanism than rule priorities (see Section 2.6). Control is specified in the form of a control program. The grammar of such programs is given in Listing 1.

A control program is interpreted during exploration of the grammer. In every state, the control program decides which rules are scheduled (i.e. allowed to be applied). A control program consists of a main control expressions, and optionally a set of function definitions. We briefly list the main features of the language.

- The smallest programming elements of a control program are the names of the rules in a grammar. Where such a name appear, only the named rule is scheduled.
- Special cases are the keywords **any** and **other**. Both serve as a kind of wildcard over the available rules: **any** executes any rule in the rule system, whereas **other** executes any rule that does not explicitly appear in the control program. For instance, if the rule system has rules a, b and c, then the control program a; **any**; b; **other**; first only allows a to be applied, then one of a, b or c, then b, and then c.
- Control expressions can be built from rules and wildcards by
  - the infix operator "|", which specifies a choice among its operands;
  - the postfix operator "*", which specifies that its operand may be scheduled zero or more times;
  - the postfix operator "+", which specifies that its operand may be scheduled one or more times;
  - The prefix operator "#", which specifies that its operand is scheduled as long as possible.

  The difference between "a*" and "#a" is that the first may optionally stop scheduling a, even if it is still applicable, whereas the latter will continue trying a until it is no longer applicable.

Listing 1: Grammar of Control Programs

```
program
        : (function|statement)*
        ;

function
        : 'function' IDENTIFIER '(' ')' block
        ;

statement
        : 'alap' block
        | 'while' '(' condition ')' 'do' block
        | 'do' block 'while' '(' condition ')'
        | 'until' '(' condition ')' 'do' block
        | 'try' block ('else' block)?
        | 'if' '(' condition ')' block ('else' block)?
        | 'choice' block ('or' block)*
        | expression
        ;

block
        : '{' statement*  '}'
        ;

condition
        : conditionliteral ('|' condition)?
        ;

conditionliteral
        : 'true' | rule ;

expression
        : expression2 ('|' expression)?
        ;

expression2
    : expression_atom ('+' | '*')?
    | '#' expression_atom
    ;

expression_atom
        : rule
        | 'any'
        | 'other'
        | '(' expression ')'
        | call
        ;

call
        : IDENTIFIER '(' ')'
        ;

rule
        : IDENTIFIER
        ;

IDENTIFIER
        : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'-'|'_')*
        ;
```

- Conditional statements allow the specification of an alternative in case certain rules do not have a matching. The conditions of **if**, **while**, **until** and **do**-**while** are restricted to a single rule name, **true**, or a choice of rules. The condition holds when at least one of the options has a match.

- The **try**-**else** statement allows more complex conditions, since the condition is incorporated in the body of the first block. In this case, the condition is true when any first possible rule (according to the block) has a match. The condition is false when the block does not lead to any rule application. For instance, the program **try** {a;b;} **else** {c;d;} goes to the second block when rule a does not have a match.

- The **alap** keyword stands for *as long as possible*. In this case, the statement is exited when — in a new iteration — the block does not lead to any rule application. Thus, **alap** has the same effect as the prefix operator #, except that it works on the level of statements rather than expresions.

- The **choice**-**do** statement has the same effect as the |-operator, except that it works on the level of statements rather than expressions.

- Functions can be fedined by the keyword **function**, followed by the function name, a pair of parentheses, and a function definition block. The parentheses are there for a possible future extension with parameters. Functions are called as expected; their semantics is defined by inlining. This means that recursive functions are not supported.

It is important to realise that control expressions are interpreted completely *deterministically*. This means that **choice** {a;b;} **or** {a;c;} has exactly the same meaning as a; **choice** {b;} **or** {c;}.

An example of a control program can be found in the control.gps grammar in the GROOVE samples.

3.6 Nested Rules

Nested rules are used to make changes to sets of sub-graphs at the same time, rather than just at the image of an existentially matched LHS. This is a quite powerful concept, which has its roots in predicate logic.

Nesting levels The specification of nested rules relies on the use of special, auxiliary nodes that stand for universal or existential quantification. These nodes are part of the rule and are connected using "in"-labelled edges. The quantifier nodes and in-edges must form a *forest*, i.e., a set of trees, within a rule; in other words, it is not allowed that a quantifier node is "in" two distinct other quantifier nodes, or that there is a cycle of quantifier nodes. Moreover, existential and universal nodes must alternate, and the root nodes must be universal. In addition, there is always an *implicit* top-level existential node, with implicit in-edges from all the explicit (universal) root nodes.

In the Editor view, the quantifier nodes are specified once more using special prefixes:

- forall: specificies a universal level: in a match of the entire rule, the sub-rule at such a level can be matched arbitrarily often (including zero times).
- forallx: specificies a *non-vacuous* universal level: in a match of the entire rule, the sub-rule at such a level must be matched at least once.
- exists: specificies an existential level: in every match of the entire rule, the sub-rule at such a level is matched exactly once.

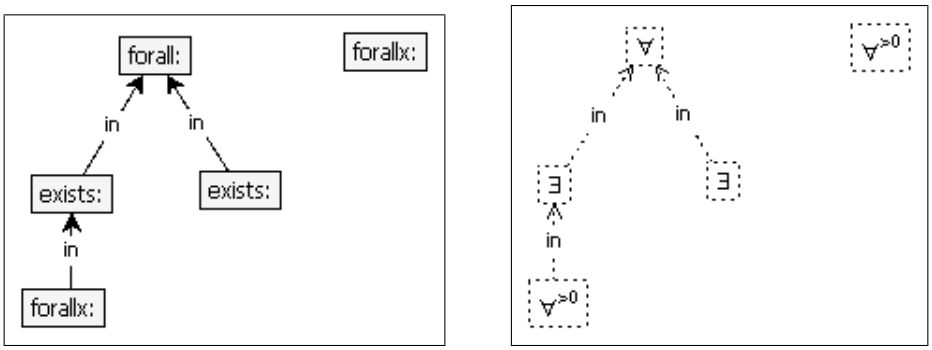
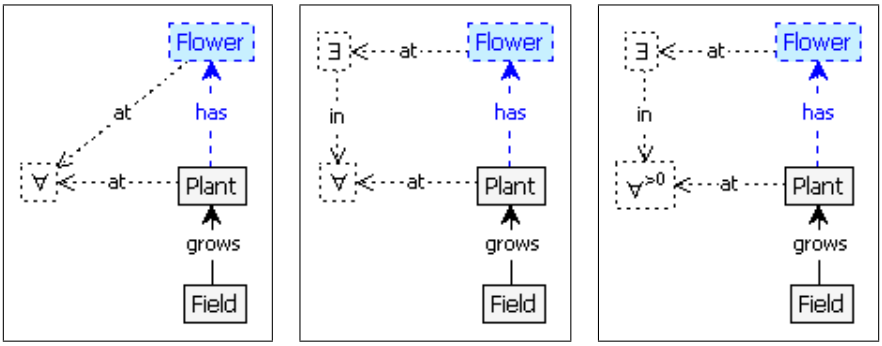


Figure 17: Edit and Display view of a nesting structure



(a) (b) (c)

Figure 18: Three different flower-picking rules.

Figure 17 gives an example of a nesting structure (leaving out the actual rule). The hierarchical structure of nesting levels corresponds to the quantifier structure of a predicate formula, where the branching stands either for conjunction (in case of universal levels), or for disjunction (in case of existentials). In other words, the structure in Figure 17 roughly reflects the predicate structure

$$\exists(\forall\exists(\forall^{>0} \vee \exists) \wedge \forall^{>0})$$

Every nesting level, represented by a quantifier node, *contains* a sub-rule. The containment relation is encoded by "at"-labelled edges from *every* node in the sub-rule to the corresponding quantifier node.

As a simple example, Rule (a) in Figure 18 will result in the removal of all Flower-labelled nodes of all Plants of a given (implicitly existentiall quantified) Field. Rule (b) is a slightly more complicated variant, which picks *exactly one* Flower of every Plant that *has at least one* Flower.

Yet another variant is given in Rule (c). Where Rule (b) is *always* applicable (as long as there is any Field-node) even if there are no flowers to be picked, Rule (c) specifies that there should be *at least one* Plant-node that can be matched — meaning that that Plant-node should have at least one Flower. This means that if a Field has Plants but none of them have Flowers, then Rule (b) matches, though its application does not change the graph, whereas Rule (c) does not match.

Another example is given in Figure 19, which specifies the rule for firing a transition in a Place-Transition net. This rule has two independent universal nodes, one to take care of the removal of tokens

from every input place of the transition to be fired, and one to take care of the addition of tokens to the output places.

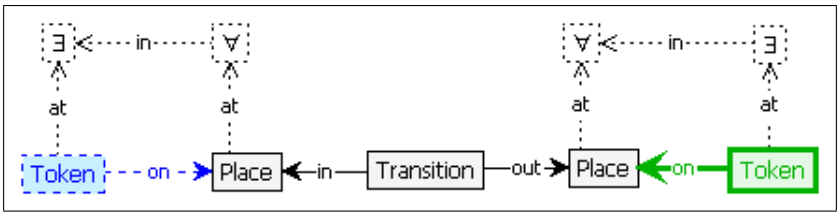


Figure 19: Petri net firing rule: One token is removed from every in-place, and one is added to every out-place.

Another example of a rule system with nested rules can be found in the copy_graph grammar in the GROOVE samples.

Named nesting levels Unfortunately, the specification of the sub-rule belonging to a given quantifier node through special at-edges fails if the sub-rule has isolated edges, since we do not support edges that start at edges. Such an isolated edge may occur if the end nodes belong to a higher nesting level.

For instance, suppose we want to specify that a girl that all boys like becomes queen. Using only at-edges, this cannot be specified. For instance, an incorrect solution is given as Figure 20.a: the applicability condition in that rule roughly corresponds to

$$\exists x : \mathsf{Girl}(x) \wedge (\forall y : \mathsf{Boy}(y) \wedge \mathsf{likes}(y, x) \Rightarrow \mathit{true})$$

meaning that the universally quantified sub-graph is trivially fulfilled. Instead, the likes-edge should be at an existential level *below* the universal, but there cannot be an at-edge starting at the likes-edge.

The solution is to use *named* nesting levels. The name of a nesting level is given as a kind of parameter in the exists- or forall-prefix: namely, the prefix becomes exists=*name*: (respectively forall=*name*:), where *name* is the (arbitrarily chosen) name of the nesting level. The edge to be associated with this label gets the same prefix. This is shown in Figure 20.b (Editor view) and c (Display view).

3.7 System Properties

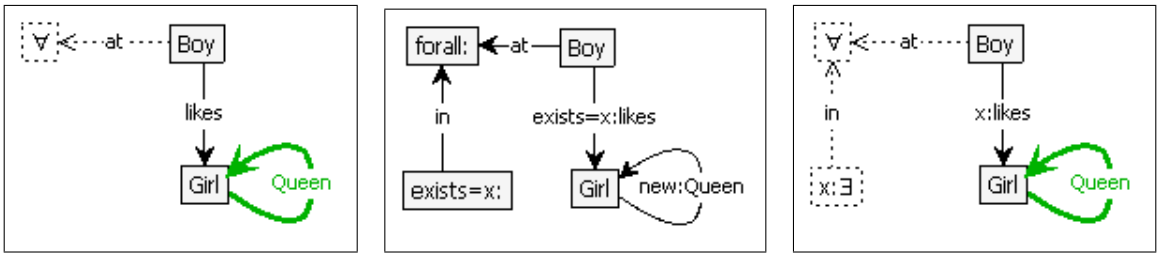
Apart from the rules, start graph and (optional) control, there are some global properties of a graph grammar. These are called the system properties. They can be set in the Simulator (through the File-menu) or by directly editing the properties file (see Section 5.3). We discuss the properties here; an opverview is provided in Table 21.

Algebra family. This property specifies the algebra to be used when interpreting data attributes — see Section 3.3. The currently supported values are:

default The default algebra, consisting of the Java types int, double, boolean and String.

point A single-point algebra, where each data type has only a single value; all constants and operations evaluate to this one value.

big An algebra where integers have arbitrary precision, and real values have a mantissa of 34 digits (to be precise, they follow the IEEE 754R Decimal128 format, as implemented by the Java type BigDecimal).



(a) (b) (c)

Figure 20: Incorrect and correct crowinging rules.

Property	Default	Meaning
remark	*empty*	One-line documentary about the rule system as a whole
matchInjective	false	Enforces injectivity of matches
algebraFamily	default	Determines which algebras are used for attributes
checkDangling	false	Makes rules inapplicable in case eraser nodes have dangling edges
checkCreatorEdges	false	Adds implicit edge embargoes for all simple edge creators
rhsIsNAC	false	Adds an implicit NAC for the entire RHS to every rule
checkIsomorphism	true	Ensures states are collapsed modulo isomorphism
transitionBrackets	false	Adds angular brackets around transition labels
transitionParameters	false	Adds parameter lists to all transition labels
controlLabels	*empty*	List of graph labels that occur rarely; used to speed up matching
commonLabels	*empty*	List of graph labels that occur frequently; used to speed up matching

Table 21: System properties overview

Match injectivity. As discussed in Section 2.4, matches are in general non-injective. By setting the matchInjective property to true, however, injectivity is enforced for all rules. In this way, GROOVE can simulate rule systems originally designed for tools that do impose injectivity always.

Dangling edge check. In general, when GROOVE deletes a node, all incoming and outgoing edges are also deleted, whether or not they were explicitly specified in the rule. This is in conformance with the so-called *SPO* (*S*ingle *P*ush*O*ut) approach. In the *DPO* (*D*ouble *P*ush*O*ut) approach, on the other hand, if a node to be deleted has an incident edge that is not explicitly deleted as well, then the rule is considered to be non-applicable. To mimic this behaviour in GROOVE, the checkDangling property should be set to true.

Creator edge check. In GROOVE, edges do not have their own identity: if an edge is added to a graph that alsready has an edge between the same nodes and with the same label, the graph actually does not change. This can be undesirable in some circumstances. By setting checkCreatorEdges to true, an implicit edge embargo is added for all creator edges; now, if an attempt is made to add an edge that is already there, the rule is inapplicable.

Treating the RHSs as NACs. There exist graph transformation applications where a graph is slowly built up but nothing is ever deleted. For instance, this holds in the important area of *model transformation*. In

such circumstances, rules should always only be applied one single time at every match; however, since nothing is deleted, the re-application of a rule can only be prevented by adding a NAC. By setting rhsIsNAC to true, such NACs are implicitly added to all rules, improving readability and maintainability of the rules.

Isomorphism check. One of the strong points of GROOVE is the fact that the graphs that it generates are compared and collapsed modulo isomorphism — meaning that there will be at most graph in the resulting state space for every isomorphism class. Though this is very effective in many modelling domains, nevertheless the isomorphism check is expensive. In case a problem being modelled is known to have little or no symmetries, so that the isomorphism check will always fail, one can set checkIsomorphism to false, thereby gaining efficiency.

Transition label formatting. The LTS view of the Simulator contains edges for all rule applications that have been explored. There are two system properties that control the way these labels are displayed.

transitionBrackets controls whether angular brackets appear around all transition labels. This option is added for backward compatibility: in previous versions, GROOVE by default showed such brackets, so if there are any applications that rely on thi sbehaviour, this property should be set to true.

transitionParameters controls whether transition labels show the value of rule parameters, is any (see Section 3.4). When set tu true, all labels will show a (possibly empty) list of parameters.

Control labels and common labels. The final pair of properties can be used to optimise the matching process, thereby improving efficiency.

controlLabels is a space-separated list of labels that do *not* occur frequently in the graph, and whose presence is a good indicator for a match at that place. When set, the matching process will start at these labels.

commonLabels is exactly the opposite: it is a space-separated list of labels that *do* occur frequently in the graph. When set, the matching process will consider these labels last.

4 Tool Usage

4.1 Overview of the Simulator (TS)

Brief, high-level overview

Panels.

Menus.

4.2 Creating and Modifying a Graph Grammar

Run-through

4.3 Exploring an LTS (IB)

Depth-First, Breadht-First, Linear, Random

4.4 Command Line Generation (IB)

Calling the Generator from the command line

4.5 Inspecting Results (AR)

Hiding and showing (parts of) graphs; Simulator options.

4.6 Exporting and Stand-alone Editing (AR)

Imager and Editor's so-far unexplained features.

4.7 Model Checking (HK)

5 I/O

Graph grammars are stored in directories with extension ".gps" (for *g*raph *p*roduction *s*ystem). Each such directory contains all information about one graph grammar, including rules, start state(s), control program (if any) and system properties, in separate files.

5.1 Graphs and rules

The input/output format used by GROOVE to store rules and graphs is GXL, which is an XML format for interchange of graphs. See http://www.gupro.de/GXL/ for information about the format, including its DTD and XSD; see also [2]. There are some conventions in the way we have used GXL to encode GROOVE-specific information.

Edge labels. Edge labels are encoded as GXL edge attributes of type String. What is actually stored is the full label, including prefixes, as seen in the Edit view. Graphs and rules are stored in precisely the same fashion, except that rules receive the extension ".gpr" (for *g*raph *p*roduction *r*ule) whereas graphs have extension ".gst" (for *g*raph *st*ate).

Graph attributes. We are using the GXL graph attributes to store some additional information about the graphs, such as the version number of the I/O-format, the fact whether it is a rule or a graph, and the priority and enabledness (if it is a rule). For the latter two see Section 2.6.

Graph layout. For every GXL file, layout information is stored in a file whose name is that of the GXL file, suffixed by the extension ".gl". For instance, the layout of the file rule.gpr is contained in rule.gpr.gl.

Layout encoding is very ad hoc, and in fact based on the way the graph-rendering library, JGRAPH, stores this information internally (see [1]). Every gl-file produced by GROOVE contains some comments briefly explaining the encoding.

5.2 Control programs

Control programs are stored in plain text in files with extension ".gcp" (for *g*raph *c*ontorl *p*rogram) within the grammar directory. If there is a file control.gcp, this is taken to be the default control program; others can be loaded in the Simulator.

5.3 System properties

System properties are stored in a file system.properties within the grammar directory. The file is a plain text file, with lines of the form "resource = value" for the different properties discussed in Section 3.7.

References

[1] JGraph: Java graph visualization and layout. URL: `http://www.jgraph.com/`, 2008.

[2] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2002.