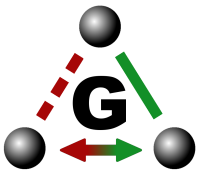
GGL Tutorial: Graph Rewrite Rules



Christoph Flamm[1,*] and Martin Mann[2]

[1] Institute for Theoretical Chemistry, Vienna University

[2] Bioinformatics Group, University of Freiburg

http://www.tbi.univie.ac.at/software/GGL/

Version April 29, 2015

Built for GGL version 4.1.1

*Send comments to xtof@tbi.univie.ac.at or mmann@informatik.uni-freiburg.de

Contents

1 Graph rewrite rules

This chapter explains how to describe a graph rewrite rule in term of the Graph Modeling Language (GML). Please note, currently only undirected graphs are supported and assumed for both the input graphs and the rewrite patterns. Theoretically, the graph grammar library is prepared to be applied on directed graphs too, but so far this was not tested nor applied!

1.1 Introduction to the specification language GML

Within the GGL, graphs and graph rewrite rules are specified in a language called GML (Graph Modeling Language). Essentially, GML is composed of hierarchical **key–value** pairs. Keys are usually strings (some identifiers) and values specify the value of the corresponding key. Values are either single values (numbers, strings, etc) or lists of key–value pairs. Lists **must always** be enclosed in opening '[' and closing ']' brackets in GML. Nesting of lists to arbitrary depth is allowed in GML. The general structure of a GML specification looks as follows:

```
key1 [
 key2 value2
 key3 [
  key4 value4
  key5 value5
 ]
 key6 value6
]
```

In the above code snippet keys 1 and 3 have both a list as value (hence the brackets). Keys 2, 4–6 are key–value pairs where the corresponding values are single values such as numbers or strings.

1.2 BNF of GML

Following is the grammar specification of GML in *Boyes Normal Form (BNF)*.

```
gml       ::= keyvalues
keyvalues ::= keyvalue (keyvalue)*
keyvalue  ::= key value
key       ::= ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']
value     ::= real | integer | string | list | operator
real      ::= sign? digit  '.' digit+ mantissa?
integer   ::= sign? digit+
operator  ::= '<' | '=' | '>' | '!'
string    ::= '"' instring '"'
list      ::= '[' keyvalues ']'
sign      ::= '+' | '-'
digit     ::= ['0'-'9']
mantissa  ::= ('E' | 'e') sign? digit+
instring  ::= ASCII-{'&', '"'} | '&' ['a'-'z''A'-'Z']  ';'
```

The GML-parser in GGL can parse any well formed GML file that conforms to the above BNF grammar specification. However the parser interprets only a subset of "known" key–value pairs (see according section) all other well-formed key–value pairs are **silently ignored** (Note: a source of errors is misspelling of known keys since the parsing is case-sensitive).

1.3 Keys for rule specification

The following table lists the relevant keys for rule specification in alphabetic order. Keys underlayed with the color gray are used to set constraints or copy-and-paste operations on vertices or edges (see section 2.4 and 3 for more details). For lists the optional enclosed keys are given in brackets.

key	**type**	**keys in list**	**comment**
constrainAdj	list	id, op, count, (edgeLabels and/or nodeLabels)	define adjacency constraints for a matched *vertex*, either nodeLabels or edgeLabels or both has to be specified
constrainEdge	list	source, target, (op), edgeLabels	define constraints for the allowed/forbidden labels for a matched *edge*
constrainNoEdge	list	source, target	define constraints that two matched vertices are *not connected* via an edge

key	**type**	**keys in list**	**comment**
constrainNode	list	id, (op), nodeLabels	define constraints for the allowed/forbidden labels for a matched *vertex*
context	list	(node), (edge)	define the *context-subgraph* of a rule
count	int	–	numeric counter for constrained rule *vertex*
copyAndPaste	list	source, id, (edgeLabels), (target)	define a copy-and-paste operation for a left-side only node. Out-edges of the source node with the given labels (or all if no specified) are copied to the target *vertex*. Optionally, the target node of the copied edges can be specified.
edge	list	source, target, label	define an *edge*.
edgeLabels	list	label	define a list of edge labels incident to a constrained *vertex*,
id	int	–	defines a numerical identifier for a *vertex*.
label	string	–	defines a textual label for a *vertex* or an *edge*.
left	list	(node), (edge), (constraintXXX)	define the *left-subgraph* of a rule. In addition several instances of constrainXXX can be added to make the rule matching more specific.
node	list	id, label	define a *vertex*

key	type	keys in list	comment
nodeLabels	list	label	define a list of *vertex* labels adjacent to a constrained *vertex*.
op	char	–	operator used in the logical expression for constraints (one of {'<', '=', '>', '!'}).
right	list	(node), (edge)	define the *right-subgraph* of a rule.
rule	list	ruleID, left, context, right, (wildcard), (copyAndPaste)	define a rule.
ruleID	string	–	define a textual name for a rule
source	int	–	define the *source-vertex* of an edge
target	int	–	define the *target-vertex* of an edge
wildcard	string	–	an optional textual label that defines which used labels for a *vertex* or an *edge* is to be matched on any other label during the left side pattern matching.

2 Chemical graph rewrite rules

In the following, the general structure of a graph rewrite rule is exemplified using the special case of instances defining chemical reactions. These can be applied to model chemical reactions based on a graph representation of molecules. Therein, molecules are defined by an undirected graph where each node represents a single atom and edges correspond to bonds of a given

valence. Within the GGL, we assume node and edge labels to be conform with the SMILES notation.

Since we are modelling chemical reactions, no atoms (i.e. nodes) are allowed to vanish or appear during the reaction. Thus, no node will be exclusively left (for vanishing nodes) or right side (appearing nodes). Label changes are possible, i.e. a node appears with different label in the left and right side of the rule. A possible reason for a label change is an altered charge of an atom as a result of the reaction.

2.1 General structure of a rewrite rule

A rewrite rule is specified with the key **rule**. Within the list value of the **rule** key 4 mandatory keys must be specified, one string valued key (**ruleID**) to name the rule and three list valued keys (**left, context, right**) defining the three subgraphs of a rewrite rule:

ruleID each rule must have a textual name which must be defined by this key.

left within the list value of this key, all edges are specified, which are broken during the chemical transformation (i.e. bonds present in the educt but absent in the product molecule(s)). Furthermore, nodes can be listed that change their label along the reaction, thus they are listed with different label within the right list. Finally, the matching can be further refined listing additional constraints.

context within the list value of this key, all nodes and edges are defined, which do not change during the chemical transformation.

right within the list value of this key, all edges are specified, which are formed during the chemical transformation (i.e. which are "new" in the product molecule(s)). Furthermore, nodes with changing label (i.e. also listed in left list) are given.

The following example illustrates how a valid rewrite rule looks like. The atoms 1-4 go into the context since chemistry is mass conserving and **no** atom can vanish or can be produced out of the blue during a chemical transformation. Note also that the total valence (total degree of each node) is preserved during the chemical reaction. Total valence preservation is a crucial feature of chemical transformations. The GML-parser in GGL checks each rule to have this property and issues an error message if this is not the case!

```
rule [
 ruleID "Double bond bromination"
 left [
  edge [ source 1 target 2 label "=" ]
  edge [ source 3 target 4 label "-" ]
 ]
 context [
  node [ id 1 label "C" ]
  node [ id 2 label "C" ]
  node [ id 3 label "Br" ]
  node [ id 4 label "Br" ]
 ]
 right [
  edge [ source 1 target 2 label "-" ]
  edge [ source 1 target 3 label "-" ]
  edge [ source 2 target 4 label "-" ]
 ]
]
```

Try to make a sketch of the above reaction and compare your result with section 4.1. (Hint: arrange the **nodes** from **context** into a polygon and draw only the vertices of the polygon on the left and right side of a reaction arrow. Insert each **edge** from **context** (if any) into both graphs left and right of the reaction arrow. Finally insert **edges** from **left** into the graph left and those from **right** into the graph right of the reaction arrow).

2.2 General steps in the derivation of writing rules

It is recommended to follow the protocol below when translating reaction mechanisms into writing rules.

1. Make a sketch of the reaction.
2. Number the atoms in the reaction mechanism.
3. Figure out which atoms/bonds are constant during the chemical transformation. (These bonds/atoms go into the **context** of the **rule**.
4. Figure out which bonds are broken during the chemical transformation. (These go into **left** of the **rule**).
5. Figure out which bonds are formed during the chemical transformation. (These go into the **right** of the **rule**).
6. Check the action of the rule on examples and counter examples to make sure that the rule does what you want.

2.3 Wildcards for atom/bond label

Generally, a graph rewrite rule has to be explicit, i.e. all node and edge labels defining the pattern to match have to be given. Sometimes, however, the specification of a dummy atom of unspecified type is more convenient to define a chemical rewrite rule to avoid and join a large number of explicit rules.

To this end, the GGL rule specification allows to define what label can be matched on any other label (applied for both nodes and edges). To this end, add the **wildcard** key-value to your rule specification. For instance, the following rule

```
rule [
 ruleID "wildcard rule"
 wildcard "myWildcard"
 left  [ node [ id 1 label "myWildcard" ] ]
 right [ node [ id 1 label "X" ] ]
]
```

would match on any node and change its label to "X". Note, the label for the wildcard can be any string as long as it is specified with the wildcard key.

Within the chemical reaction encoding, per default the wildcard label **"*"** is defined. It can be used for both atom and bond label specification and matches any other label. Note, this wildcard label is fixed and hardcoded and cannot be changed within the chemical framework.

The use of wildcards within rules broadens their applicability but might result in too general patterns. To this end, additional constraints might be needed that restrict the generality of the wildcard usage. The currently supported constraints are discussed in the following.

Note, wildcards are also allowed within some constraints and copy-and-paste operations as discussed in the following.

2.4 Constraining atoms/bonds

To simplify rule formulation or to make rules more specific, it is sometimes necessary to further constrain atoms or bonds (of the to-be-matched rule's left side). In the following, the available constraints are exemplified.

2.4.1 Constrain allowed atom labels

If an atom label is not explicitly specified using the wildcard character "*" but only a specific set of atom labels should be allowed, a node label constraint has to be set. This can be done use the **constrainNode** statement.

For instance, the following constraint restricts the allowed labels for the atom with node id 1 to carbon (C) or nitrogen (N). Otherwise, the node could have been matched with any atom within a molecule.

```
...
context [
  ...
  node [ id 1 label "*" ]
  ...
]
left [
  constrainNode [ id 1 op = nodeLabels [ label "C" label "N" ] ]
]
...
```

We can achieve the inverse result when changing the operator to `op !` which makes the given node labels the set of *forbidden* labels. Thus, the contraint would enforce that the matched node shows *none* of the given labels.

2.4.2 Constrain allowed bond labels

As for atom labels, the wildcard character "*" can be used as an edge label to enable a general matching definition. One can define constraints in a similar way to restrict the allowed bond labels using the **constrainEdge** statement.

For instance, the following constraint restricts the allowed labels for the bond between the atoms with id 1 and 2 to be a single bond ("–") or a double bond ("=").

```
...
context [
  ...
  edge [ source 1 target 2 label "*" ]
  ...
]
left [
  constrainEdge [ source 1 target 2 op =
    edgeLabels [ label "-" label "=" ]
  ]
]
...
```

As for the node label constraint, we can achieve the inverse result when changing the operator to `op !` which makes the given edge labels the set of

forbidden labels. Thus, the contraint would enforce that the matched edge shows *none* of the given labels.

Note, this constraint is only usefull if either *no multiple parallel edges* are possible/present between the constrained source and target nodes or if all parallel edges between these two nodes are to be constrained.

2.4.3 Forbid a certain bond

Since subgraph isomorphism focuses on the matching of present nodes and edges, some graph rewrite rules need to explicitely state a non-existance of a certain edge. This can be done using the **constrainNoEdge** statement as exemplified in the following for the nodes with id 1 and 2.

```
...
constrainNoEdge [ source 1 target 2 ]
...
```

2.4.4 Constrain adjacency (or degree)

Often it is of interest to constrain the adjacent nodes and edges for a given node. Using **constrainAdj** a sophisticated adjacency restriction can be set. It is based either on a given list of node or edge labels or a combination. For each constraint the number of nodes/edges matching the given labels is determined and evaluated according to the given operator and targeted value.

For instance the following constrain enforces that atom with node id 1 has at least 3 adjacent single bonds (edges with label "–").

```
...
constrainAdj [ id 1 op > count 2 edgeLabels [ label "-" ] ]
...
```

In order to specify that any edge or node label can be matched either a wildcard label can be specified or the according list can be ommitted. Thus, if no node or edge labels are given, all nodes/edges are taken into consideration. If both node *and* edge labels are specified, only edges are counted where the edge label *and* the targeted node label are among the allowed labels.

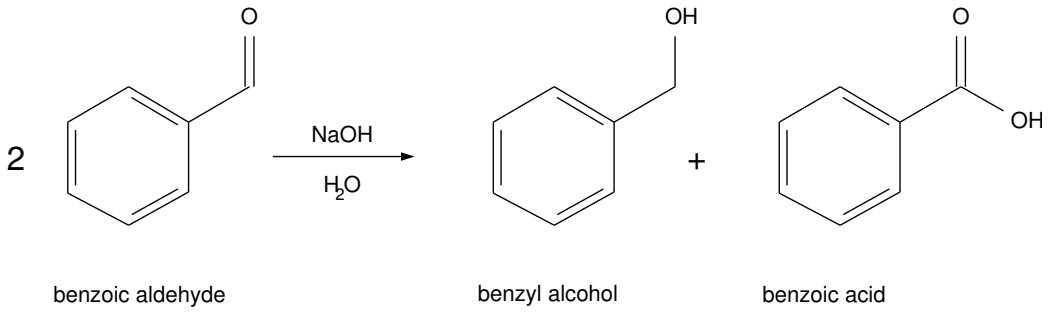
Therefore, a *degree constraint* can be simply expressed using

```
constrainAdj [ id 1 op = count 2 ]
```

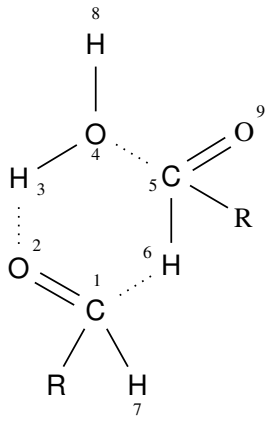
Note: all according nodes and edges are taken into account, i.e. also the nodes/edges that are *explicitely* stated within the rule.

A good example to illustrate adjacency constraints is the Cannizzaro reaction. The reaction involves the base-induced disproportionation (i.e. the

self oxydation-reduction reaction) of an aldehyde lacking a hydrogen atom in the α-position of the carbonyl-group yielding a 50:50 product mixture of the corresponding alcohole (reduction product) and carboxylic acid (oxydation product).



Let us assume that the canizzarro reaction proceeds *via* a cyclic six-membered *"imaginary transition state" (ITS)* (arranging 2 aldehydes and 1 water molecule)



than the following bond changes happen

- broken bonds: 1–2 (C=O), 3–4 (H−O) and 5–6 (C−H).
- formed bonds: 1–2 (C−O), 2–3 (H−O), 4–5 (O−C) and 6–1 (H−C).
- constant: atoms 1–9 and bonds 1–7 (C−H), 4–8 (O−H) and 5–9 (C=O).

resulting in the rewriting rule

```
rule [
 ruleID "cannizzaro reaction too general"
 context [
  node [ id 1 label "C" ]
  node [ id 2 label "O" ]
  node [ id 3 label "H" ]
  node [ id 4 label "O" ]
  node [ id 5 label "C" ]
  node [ id 6 label "H" ]
  node [ id 7 label "H" ]
  node [ id 8 label "H" ]
  node [ id 9 label "O" ]
  edge [ source 1 target 7 label "-" ]
  edge [ source 4 target 8 label "-" ]
  edge [ source 5 target 9 label "=" ]
 ]
 left [
  edge [ source 1 target 2 label "=" ]
  edge [ source 3 target 4 label "-" ]
  edge [ source 5 target 6 label "-" ]
 ]
 right [
  edge [ source 1 target 2 label "-" ]
  edge [ source 2 target 3 label "-" ]
  edge [ source 4 target 5 label "-" ]
  edge [ source 6 target 1 label "-" ]
 ]
]
```

The above rule is very general and matches any aldehyde regardless what R actually is. However, aldehydes possessing a hydrogen at the atom adjacent to the carbonyl group (e.g. R = CH_3) form the enol tautomer under basic conditions and cannizzaro reaction is **not** observed. To make the cannizzaro rule specific for aldehydes without a hydrogen in the α position of the carbonyl group, we first have to add two C atoms (10, 11) and the respective bonds (1–10, 5–11) to the *context subgraph* and disallow hydrogens on atoms 10 and 11 by using a **constrainAdj** statement.

```
rule [
 ruleID "cannizzaro restrictive"
 context [
  node [ id 1 label "C" ]
  node [ id 2 label "O" ]
  node [ id 3 label "H" ]
  node [ id 4 label "O" ]
  node [ id 5 label "C" ]
  node [ id 6 label "H" ]
  node [ id 7 label "H" ]
  node [ id 8 label "H" ]
  node [ id 9 label "O" ]
  node [ id 10 label "C" ]
  node [ id 11 label "C" ]
  edge [ source 1 target 7 label "-" ]
  edge [ source 4 target 8 label "-" ]
  edge [ source 5 target 9 label "=" ]
  edge [ source 1 target 10 label "-"]
  edge [ source 5 target 11 label "-"]
 ]
 left [
  edge [ source 1 target 2 label "=" ]
  edge [ source 3 target 4 label "-" ]
  edge [ source 5 target 6 label "-" ]
  constrainAdj [ id 10 op = count 0 nodeLabels [ label "H" ] ]
  constrainAdj [ id 11 op = count 0 nodeLabels [ label "H" ] ]
 ]
 right [
  edge [ source 1 target 2 label "-" ]
  edge [ source 2 target 3 label "-" ]
  edge [ source 4 target 5 label "-" ]
  edge [ source 6 target 1 label "-" ]
 ]
]
```

2.5 Radicals

A radical is an atom that has unpaired valence electrons or an open electron shell, and therefore may be seen as having one or more "dangling" covalent bonds. The GGL chemistry framework "sanity checks" for rules and produced molecules do not allow such atoms. Nevertheless, one can still represent radicals using a simple trick following the observation that radicals are usually only intermediates of reactions and thus both created and destroyed

by a chemical reaction part of the reaction set applied. Therefore, radicals are represented by atoms with according additional charge information *plus* a radical specific class name. The latter ensures the distinction of radicals encoded in such a way from normal atoms with the same charge. These specific radical labels are than used in the "destruction" reaction to replace the radical with the according atom label.

A simple example is the reaction $Cl_2 \rightarrow Cl\bullet + Cl\bullet$ where chlorine gas is broken down by ultraviolet light to atomic chlorine radicals. This can be expressed using

```
rule [
 ruleID "chlorine gas to radical"
 context [ ]
 left [
  node [ id 1 label "Cl" ]
  node [ id 2 label "Cl" ]
  edge [ source 1 target 2 label "-" ]
 ]
 right [
  node [ id 1 label "Cl-:1" ]
  node [ id 2 label "Cl-:1" ]
 ]
]
```

where `Cl-:1` represents the chlorine radicals.

Note: You have to ensure, that the class identifier used to encode for radicals (here 1) is not used for other class descriptions.

2.6 Group placeholders within rules

The specification of (bio)chemical reactions often requires the representation of large (unchanged) parts of molecules in order to make the rule as specific as the chemical reaction. A classic example is the involvement of helper molecules like ATP, NADH, etc. that are only slightly changed but have to be represented completely to avoid the application of the rule using similar molecules.

To this end, the `GGL` supports the specification of molecular groups as pseudo-atoms within chemical rule definitions. They allow for a much easier and compact rule definition and avoid potential typos and mistakes.

As an example consider the lactat-dehyrogenase from the citrat-cycle given by NAD^+ + lactate $\rightarrow$ NADH + pyruvate. NADH is a large molecule comprising 66 atoms. Thus, a complete specification would require the definition of all NADH atoms and bonds together with the according parts

of lactate and pyruvate incorporating 76 atoms in total. Furthermore, this would be the case for all other NADH-dependent reactions as well.

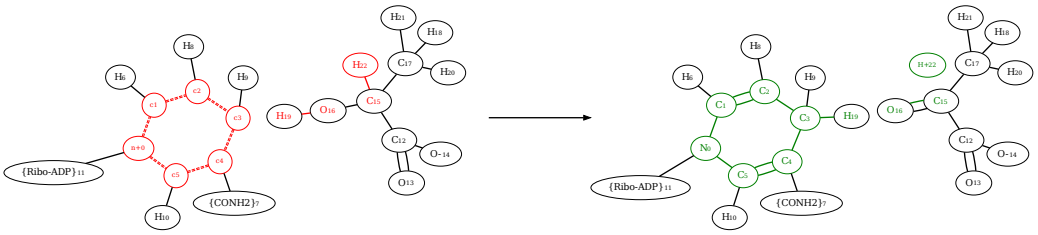


Figure 1: Lactat-dehydrogenase : NAD^+ + lactate $\rightarrow$ NADH + pyruvate. The picture exemplifies the use of group identifiers to compact the rule specification. The colors indicate if specified as context (black), left (red), or right (green). Note, such a representation reduces the rule specification from 76 to only 23 atoms.

Using group identifiers, the definition of the lactat-dehydrogenase becomes much more compact as exemplified in Fig. 1. With only 23 atoms, the whole reaction is described. Note, the rule specification uses two group descriptors. Each is replaced during the rule `GML` parsing with according molecule components/subgraphs, i.e. `{CONH2}` is replaced with a $CONH_2$ group and `{Ribo-ADP}` with a ribose and attached adenosine.

Each group shows as interface exactly one proxy node that will replace the pseudo atom labeled with the group ID. Thus, a rule can only change bonds with the proxy node, the rest of the group is statically added to the rule context. It is possible to specify label changes of the proxy node atom but these are restricted to charge changes as exemplified below. An explicit change of the proxy node label (e.g. make it aromatic "`C`" $\rightarrow$ "`c`") is not possible. Further information on how molecular groups have to be defined etc. are given in the according tutorial "`GGL` Tutorial: Molecular Groups".

```
...
left [
  node [ id 1 label "{GROUP}" ]
  ...
]
right [
  node [ id 1 label "{GROUP}+" ]
  ...
]
...
```

2.7 Visualization of chemical rules

The GML definition of chemical rules can become quite large and hard to read. To ease their creation and to allow for a simple evaluation, the GGL sports the visualization script `chemrule2svg.pl` within its Perl module.

Given a chemical reaction in GML notation, the script produces a graphical depiction in Scalable Vector Graphics (SVG) format. Therein, a color coding is used to highlight what parts are defined in the context (black), left (red), or right (green) part of the rule. An example is given in Fig. 1. The `chemrule2svg.pl` script uses the `OpenBabel` package to create the 2D depictions of the molecules and thus requires its presence.

3 Copy-and-Paste operations

Some graph operations require the deletion of one or several nodes but need to maintain and copy the former connectivity of the nodes to be removed. In such cases, a copy-and-paste operation can be used, specified by the list keyword **copyAndPaste**. Within the GML specification the source node to be deleted (has to be a left-side only node) and a target node to inherit the connectivity of the source (has to be a non-left node) are specified. Optionally, a set of edge labels to constrain the edges to copy can be specified. If no edge label list is given or the wildcard is among the labels, all edges will be copied. The edges to copy can be further specified by giving the target node of the edges of interest.

A small example is given in the following:

```
rule [
 ruleID "copy-and-paste"
 left    [ node [ id 1 label "A" ] ]
 context [ node [ id 2 label "B" ] ]
 right   [ node [ id 3 label "C" ] ]
 copyAndPaste [ source 1 id 2 ]
 copyAndPaste [ source 1 id 3 edgeLabels [ label "-" ] ]
 copyAndPaste [ source 1 id 3 edgeLabels [ label "-" ] target 2 ]
]
```

Within the example, node A is deleted. But beforehand, all out-edges of A are copied to node B. In addition, all out-edges of A with the edge label "–" are copied to the newly created node C. Note, using copy-and-paste operations it is possible to duplicate edges if needed. This is exemplified with the third copy-and-paste operation where all edges between node A and B with edge label "–" are again copied to be edges between node C

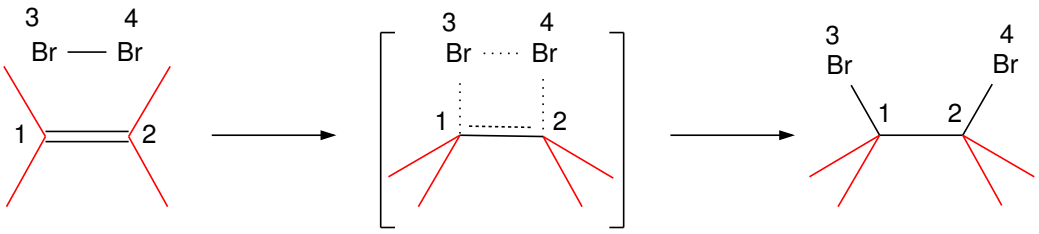
and B, thus if existing two such edges are created in combination with the second copy-and-paste operation.

Note, copy-and-paste operations are based on the left side pattern matching, i.e. the edges copied are *without* the edges to add from the right-side and *including* the edges from the left-side of the rule specification.

4 Examples

4.1 Bromination of a double bond

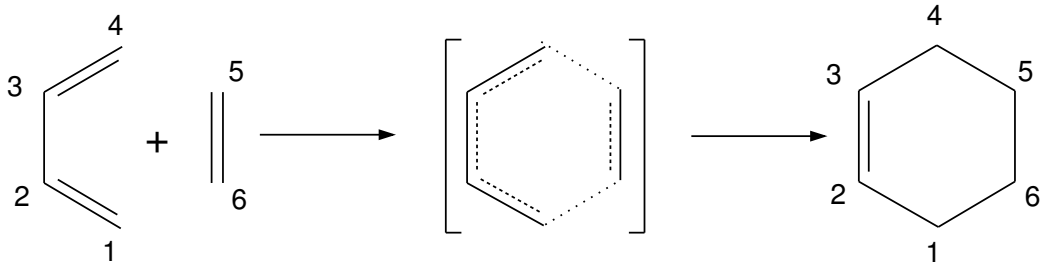
In this reaction a Br_2 molecule is added to a C=C bond. The reaction is thought to go *via* a 4-cyclic transition state (bracketed structure).




```
rule [
 ruleID "Double bond bromination"
 left [
  edge [ source 1 target 2 label "=" ]
  edge [ source 3 target 4 label "-" ]
 ]
 context [
  node [ id 1 label "C" ]
  node [ id 2 label "C" ]
  node [ id 3 label "Br" ]
  node [ id 4 label "Br" ]
 ]
 right [
  edge [ source 1 target 2 label "-" ]
  edge [ source 1 target 3 label "-" ]
  edge [ source 2 target 4 label "-" ]
 ]
]
```

4.2 Diels-Alder reaction

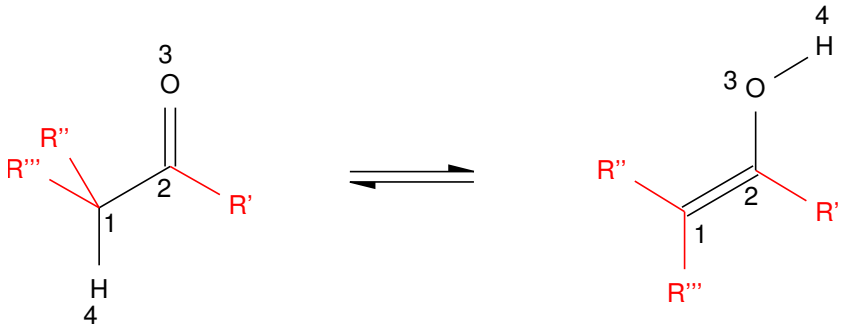
The Diels-Alder reaction is a [4+2]-cycloaddition between a conjugated diene and an alkene, commonly termed the dienophile, to form a (substituted) cyclohexene system.




```
rule [
 ruleID "Diels-Alder reaction"
 left [
  edge [ source 1 target 2 label "=" ]
  edge [ source 2 target 3 label "-" ]
  edge [ source 3 target 4 label "=" ]
  edge [ source 5 target 6 label "=" ]
  constrainNoEdge [ source 1 target 5 ]
  constrainNoEdge [ source 4 target 6 ]
 ]
 context [
  node [ id 1 label "C" ]
  node [ id 2 label "C" ]
  node [ id 3 label "C" ]
  node [ id 4 label "C" ]
  node [ id 5 label "C" ]
  node [ id 6 label "C" ]
 ]
 right [
  edge [ source 1 target 2 label "-" ]
  edge [ source 2 target 3 label "=" ]
  edge [ source 3 target 4 label "-" ]
  edge [ source 4 target 5 label "-" ]
  edge [ source 5 target 6 label "-" ]
  edge [ source 6 target 1 label "-" ]
 ]
]
```

4.3 Keto-enol isomerization

The keto-enol isomerization refers to a chemical equilibrium between a keto form (a ketone or an aldehyde) and an enol. The enol and keto forms are said to be tautomers of each other. The interconversion of the two forms involves the movement of a proton and the shifting of bonding electrons.



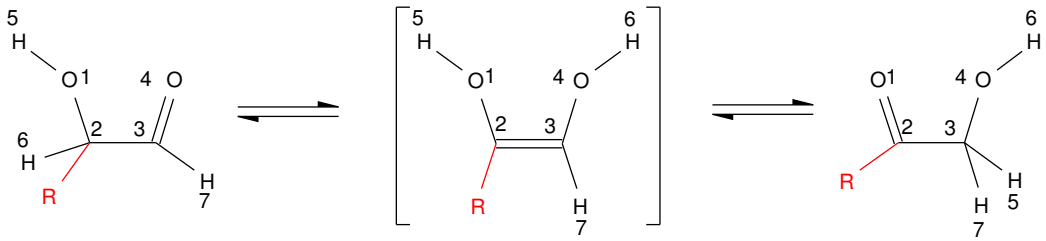

```
rule [
 ruleID "Keto-enol isomerization forward"
 left [
  edge [ source 1 target 4 label "-" ]
  edge [ source 1 target 2 label "-" ]
  edge [ source 2 target 3 label "=" ]
  constrainAdj [ id 2 op = count 1 nodeLabels [ label "O" ] ]
 ]
 context [
  node [ id 1 label "C" ]
  node [ id 2 label "C" ]
  node [ id 3 label "O" ]
  node [ id 4 label "H" ]
 ]
 right [
  edge [ source 1 target 2 label "=" ]
  edge [ source 2 target 3 label "-" ]
  edge [ source 3 target 4 label "-" ]
 ]
]
```

```
rule [
 ruleID "Keto-enol isomerization backward"
 left [
  edge [ source 1 target 2 label "=" ]
  edge [ source 2 target 3 label "-" ]
  edge [ source 3 target 4 label "-" ]
  constrainAdj [ id 2 op = count 1 nodeLabels [ label "O" ] ]
 ]
 context [
  node [ id 1 label "C" ]
  node [ id 2 label "C" ]
  node [ id 3 label "O" ]
  node [ id 4 label "H" ]
 ]
 right [
  edge [ source 1 target 4 label "-" ]
  edge [ source 1 target 2 label "-" ]
  edge [ source 2 target 3 label "=" ]
 ]
]
```

(Note that atom 2 is constrained to has only one adjacent oxygen atom. This is done to exclude carboxyl groups (CO_2H) from enolization.)

4.4 Aldose-Ketose transformation

This reaction from carbohydrate chemistry, also known under the name **Lobry de Bruyn van Ekenstein transformation**, is the base or acid catalyzed transformation of an aldose into the ketose isomer or *vice versa*. The transformation is thought to go *via* a tautomeric enediol (bracketed structure) as reaction intermediate.



Since the reaction is reversible we have to put the forward and backward reaction into the rule file.

```
rule [
 ruleID "Aldose-ketose forward"
 left [
  edge [ source 1 target 2 label "-" ]
  edge [ source 1 target 5 label "-" ]
  edge [ source 2 target 6 label "-" ]
  edge [ source 3 target 4 label "=" ]
  constrainAdj [ id 2 op = count 1 nodeLabels [ label "O" ] ]
 ]
 context [
  node [ id 1 label "O" ]
  node [ id 2 label "C" ]
  node [ id 3 label "C" ]
  node [ id 4 label "O" ]
  node [ id 5 label "H" ]
  node [ id 6 label "H" ]
  node [ id 7 label "H" ]
  edge [ source 2 target 3 label "-" ]
  edge [ source 3 target 7 label "-" ]
 ]
 right [
  edge [ source 1 target 2 label "=" ]
  edge [ source 3 target 4 label "-" ]
  edge [ source 3 target 5 label "-" ]
  edge [ source 4 target 6 label "-" ]
 ]
]
```

```
rule [
 ruleID "Aldose-ketose backward"
 left [
  edge [ source 1 target 2 label "=" ]
  edge [ source 3 target 4 label "-" ]
  edge [ source 3 target 5 label "-" ]
  edge [ source 4 target 6 label "-" ]
  constrainAdj [ id 2 op = count 1 nodeLabels [ label "O" ] ]
 ]
 context [
  node [ id 1 label "O" ]
  node [ id 2 label "C" ]
  node [ id 3 label "C" ]
  node [ id 4 label "O" ]
  node [ id 5 label "H" ]
  node [ id 6 label "H" ]
  node [ id 7 label "H" ]
  edge [ source 2 target 3 label "-" ]
  edge [ source 3 target 7 label "-" ]
 ]
 right [
  edge [ source 1 target 2 label "-" ]
  edge [ source 1 target 5 label "-" ]
  edge [ source 2 target 6 label "-" ]
  edge [ source 3 target 4 label "=" ]
 ]
]
```