The AGG 1.4.0 Development Environment
The User Manual

e-mail: {agg-team}@tfs.cs.tu-berlin.de,
http://tfs.cs.tu-berlin.de/agg

Contents

Introduction

This renewed AGG manual corresponds to the AGG version 1.2.5 and latter. The AGG version 1.2.4 is mainly covered by this manual, too. The installation instructions of the AGG tool can be found in a README file of the AGG package that you can download from URL `http://tfs.cs.tu-berlin.de/agg`. The AGG package also contains graph grammar examples, two of which we will use for our explanations.

The aim of this manual is to give a short but sufficient knowledge to understand how to specify a graph grammar:

1. to define and validate a set of transformation rules,
2. to perform transformations of the host graph,
3. to analyze the specified graph transformation system.

The AGG language is a rule based visual language supporting an algebraic approach to graph transformation. It aims at specifying and rapid prototyping applications with complex, graph structured data. The AGG environment is designed as a tool to edit directed, typed and attributed graphs and to define a graph grammar (i.e. a start (host) graph plus a set of transformation rules) as input for the graph transformation engine of the system. Having an AGG graph grammar at hand, it may be validated using AGG's analysis techniques, namely critical pair analysis, consistency checking and termination criteria for Layered Graph Transformation Systems (LGTS).

The main characteristics of the AGG language can be described as follow:

- Complex data structures are modeled as graphs which may be typed over a type graph.
- AGG graphs may be attributed by Java objects and types. Basic data types as well as object classes already available in Java class libraries may be used.
- Moreover, new Java classes may be included.
- The system's behavior is specified by graph rules using an if-then description style.
- Moreover, AGG features rules with *Negative Application Conditions* to express requirements for non-existence of substructures.
- The graph rules may be attributed by Java expressions which are evaluated during rule applications.
- Rules may have attribute conditions being boolean Java expressions.
- Application of a graph rule transforms the structure graph.
- Application of several rules sequentially shows an application scenario.

- A consistency control structure can be defined as global graph consistency conditions (constraints). They describe invariants on graphs. Additionaly, these global consistency conditions can be transformed into *Post Application Conditions* for an individual rule.

With AGG it is possible to define typed attributed graph transformation with *node type inheritance*. This means that the attributed type graph can be enriched by an inheritance relation between nodes. Each node type can have only one direct ancestor from which it inherits the attribute and edge types. Rules using this feature are equivalent to a number of concrete rules, resulting from the substitution of the ancestor nodes by the nodes in their inheritance clan. Therefore, rules become more compact and suitable for their use in combination with object-oriented modelling.

The AGG language and design concepts are described in

- "Introduction to the Language Concepts of AGG" by Michael Rudolf and Gabriele Taentzer (`http://tfs.cs.tu-berlin.de/agg/intro.ps.zip`).
- "Concepts and Implementation of an Interpreter for Attributed Graphtransformation" Diploma thesis of Michael Rudolf (in german) (`http://tfs.cs.tu-berlin.de/agg/Diplomarbeiten/Rudolf.ps.gz`).
- "Design and Implementation of an Attribute Manager for Conditional and Distributed Graph Transformation" Diploma thesis of Boris Melamed (`http://tfs.cs.tu-berlin.de/agg/Diplomarbeiten/Melamed.ps.gz`).

The AGG environment provides graphical editors for graphs, rules and graph consistency constraints and an integrated textual editor for Java expressions. Moreover, visual interpretation and step by step transformation of attributed graph grammars is supported. While step means performing direct derivations (transformation) for user-selected productions (rules) and occurrences (matches), a whole transformation sequence is executed in the interpretation mode. Furthermore, analysis tools for graph transformation systems are available.

In the following, first the editing and afterwards the interpretation and validation facilities are presented for the sample applications.

1 Editing

The editing environment of the AGG system provides a comprehensive functionality for the input and modification of attributed graph grammars with a mouse/menu-driven user interface.
There are different editors:

- an editor for node and edge types with the possibility to name and set a simple graphical layout for each node and edge type,

- a graphical editor for graphs (host graphs and type graphs),
- a graphical editor for rules, that supports the editing of rules consisting of a left and a right hand rule side and (optionally) one or more negative application conditions (NACs),
- a textual editor for Java expressions which can be used as attributes,
- a tree view window to edit graph grammars and name its ingredients,
- a textual editor to order rules in layers.

1.1 The AGG Graphical User Interface

We present the editing and the interpretation facilities for the sample application *ShortestPath*.
In Figure 1 you can see the graphical user interface of the AGG system. To the left, the window with the current graph grammars is shown. It is possible to have more than one graph grammar loaded. This is visualized by a GraGra tree, the current graph grammar, start graph or rule being highlighted.
In Figure 1, there is only one graph grammar, namely *ShortestPath.ggx* consisting of the start graph *Cities* and seven rules (five of them having NACs). In the GraGra tree window the names of graph grammars, start graphs, rules and NACs can be edited and the rule or graph to be edited or used for transformation steps can be selected here by clicking on its icon. The selected graph or rule is then shown in its graphical editor. The graph editor can be found in the lower right part, while the rule editor is situated in the upper right part exact by above the graph editor.
The File menu allows to create, load, save, export and import a graph grammar and to exit the AGG system. Each graph grammar consists of one start graph and a number of rules. The menus Edit and Mode allow to edit graphs, rules and attributes. The Transform menu is visited for interpreting and debugging graph transformation. Here, the definition of a match and the execution of a transformation step or sequence is possible. The Parser menu may be used to activate and start graph parsing processes. The Analyzer menu allows to apply some kinds of analysis technique to a grammar. There are critical pair analysis, consistency check for graphs and termination checks available for Layered Graph Transformation Systems. The Preferences menu should be used to set options for transformation, parsing and critical pair analysis. The Help menu provides short description of AGG GUI.
Figure 1 shows the first rule *start* of our *ShortestPath* example in the rule editor and the result of the *start* rule having been applied to the initial road map of the example.
Now we describe how to create and delete graph grammars and their components, how to define types for graph objects, how to draw nodes and edges in the graphical editors, how to edit rules, and how to add attributes making use of the textual attribute editor.

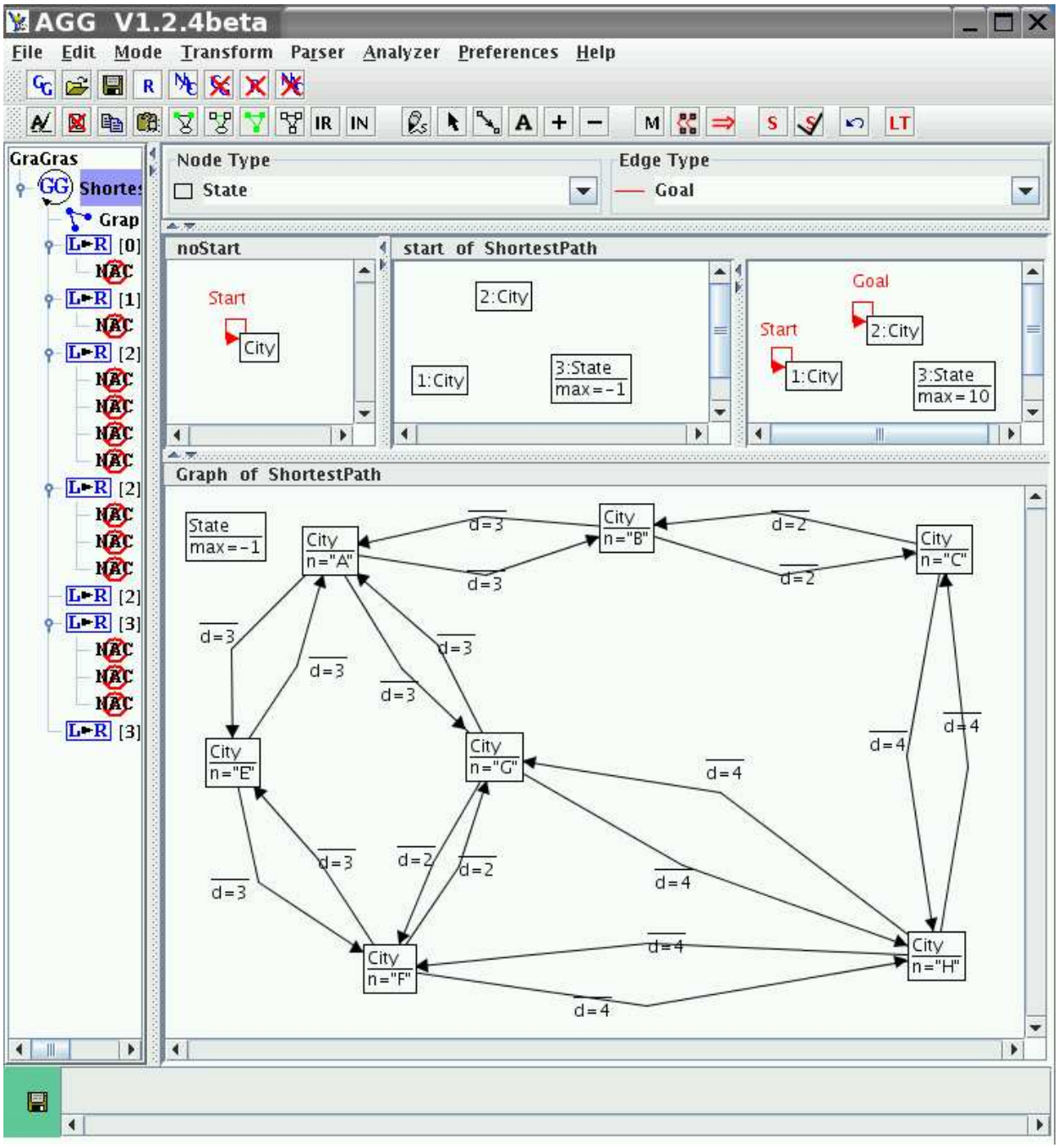


Figure 1: The AGG Graphical User Interface

1.2 Creating and Deleting Graph Grammars and their Components

In menu File, a new graph grammar may be created by clicking on menu entry New GraGra. A new graph grammar named *GraGra* is created, together with the start graph named

Graph and one rule named *Rule* by default. These new components are shown in the tree view window where the default names can be changed by clicking on the name text field twice. The default edit mode is the *Draw* mode, and the new empty start graph and the empty rule *Rule*0 have been loaded into their graphical editors where graphs may be drawn. Using pop-up menus one can modify the structure of a grammar elements and obtain their functionalities.
To create a new rule for the currently selected graph grammar, the pop-up menu of GraGra has to be used choosing item New Rule. To create a new negative application condition (NAC) for the currently selected rule, the pop-up menu of Rule has to be used choosing New NAC item. Such pop-up menus will appear after pressing the right mouse button when the cursor points to the corresponding node of the grammar tree. Again, the newly created items are added to the tree view where they can be renamed or selected for editing.
A graph grammar is closed by selecting item Close in menu File while the graph grammar to be closed is selected (highlighted in the tree view) or using the pop-up menu of GraGra.
A rule or a NAC is deleted by selecting the item Delete of the appropriate pop-up menu.
A grammar may contain a type graph. It can be created by selecting the item New Type Graph of the pop-up menu of GraGra. The pop-up menu of type graph is visited to decide how the type graph should be used during editing graphs and applying rules.
Moreover, a grammar may contain some atomic graph constraints and logical constraints (formulae) based on atomic graph constraints. The pop-up menu of GraGra allows creating and checking such components and their own pop-up menus - editing and deleting.

1.3 Defining Types for Graph Objects

The type editor for node and edge types can be found above the rule editor. In the beginning, no types are defined and at least one node and one edge type have to be defined to draw nodes and edges. The type editor has a limited set of elementary graphics to represent nodes and edges. Nodes may be shaped as rectangles with / without rounded corners, ovals and circles. Edges may be solid, dotted or dashed lines. Both nodes and edges may have different colours. A type name may be added to identify a graph object's type but is not necessary. For example, there might be defined the node types *red rectangle*, *blue rectangle*, *red circle with type name* City and *red rectangle with type name* City. These are four different node types. The type of an object has to be defined before actually drawing the object. A type is defined by clicking on the graphical Node Type (or Edge Type) symbol and selecting a graphical shape and/or colour in the context menu of colours and shapes/styles. Then the type name must be entered into the Node Type or Edge Type text field. Even if the type name is left empty, at least the return key must be pressed in the Node/Edge Type text field. Then, the new type is stored in the Node Type or Edge Type list. It can be accessed by mouse click whenever a new object of this type is to be drawn. The current type is indicated by the graphical symbol next to the Node/Edge Type text field and the name inside the text field. In the graphical editors, the type name (if any) is shown inside the node shape or next to the middle point of an edge.
The redefinnition of a node or an edge type is possible by clicking on the graphical Node Type (or Edge Type) symbol and selecting item Redefine. Now, the name or graphical

representation of current type may be changed. At least the return key must be pressed in the text field to accept the type changes. NOTE: Do not forget to deselect item Redefine, if you want create a new node/edge type.
A node or an edge type can be deleted by clicking on the graphical Node Type (or Edge Type) symbol and selecting item Delete. If current type was already used for typing, a warning dialog appears to ask you for confirmation since all objects of this type and possibly adjacent edges will be deleted, too. Otherwise, the type will be deleted immediately.

1.4 Defining a Type Graph

AGG supports the possibility to create a *typed graph grammar*. In a *typed graph grammar* the type description by type sets (label sets) for nodes and edges is extended by a *type graph*. One major disadvantage of type sets is that edge types do not prescribe the types of source and target nodes. This additional information can be given by *type graphs*: A fixed graph TG, called *type graph*, represents the type information, and the typing of a graph G over TG is given by a total graph morphism $t : G \rightarrow TG$.
In general, nodes and edges of TG represent nodes and edges types, while attributes are attribute declarations. The edges in TG represent a structural relationship among objects. A node type can be compared to a class in UML class diagram and an edge type can be compared to an association. It is important to state how many objects may be connected through an instance of an edge type. This "how many" is called *the multiplicity* of an association's role in UML, and is specified by a range of values or an explicit value.
When you state a *multiplicity* at the target end of an edge type, you specify the number of nodes which may be connected to one source node across edges of the given edge type. A *multiplicity* at the source end of an edge type is interpreted similarily. It is also possible to state a *multiplicity* at a node.

Using *typed graph grammar* has several positive effects since multiplicities pose additional graph constraints on a graph:

- the number of negative application conditions for rules might decrease,
- the number of consistency conditions might decrease,
- the efficiency of the critical pair analysis might increases, since a number of overlapping graphs might be ruled out according to multiplicities given. Especially upper bounds of multiplicites are useful to reduce the number of overlapping graphs.

A new *type graph* can be created by item New Type Graph of the pop-up menu of GraGra. An empty type graph will be loaded in the graph editor. Editing this graph is similar to editing the host graph of a grammar.
Using the pop-up menu of TypeGraph shown in Figure 2 we can define the usage of the *type graph*.
The meaning of the items is :

- disabled - the type graph is ignored. Thus, all graphs contain objects with types defined in the type set of the grammar. Multiplicities are also ignored.

Figure 2: Pop-up menu of TypeGraph

- enabled - the type graph is basically used. Thus, all graphs only contain objects with types defined in the type graph, but multiplicities are not checked.
- enabled with max - the type graph is basically used. Thus, all graphs only contain objects with types defined in the type graph. Multiplicities in all graphs should satisfy the defined maximum bounds.
- enabled with min and max - the type graph is used completely. Thus, all graphs only contain objects with types defined in the type graph. Multiplicities in all graphs must satisfy the defined maximum bounds and additionally the host graph must satisfy the defined minimum bounds.
- Delete - the type graph will be destroyed and thus not active anymore.

If the type graph check is set to *enabled*, *enabled with max* or *enabled with min and max*, the defined type graph is used during all changes of the grammar.
The *multiplicity* of a node type or of the source and target of an edge type can be set in a multiplicity dialog which is accessed by the pop-up menu Operations of nodes and edges when you are editing a type graph.

1.4.1 Type Graph With Node Type Inheritance

AGG gives us a possibility to enrich the type graph with an inheritance relation between nodes. Each node type can have only one direct ancestor (parent) from which it inherits the attributes and edges. The parent node can be defined using item Set Parent of the pop-up menu Operations of nodes. After selecting the parent by clicking the left button on the appropriate node the inheritance edge appears. Deletion inheritance relation is possible using item Unset Parent of the pop-up menu Operations of nodes.
Please note: The *multiplicity* of a parent type also has an effect on the number of instances of successor types in a graph. Since all successor types can be seen as instances of that parent type, the sum of all instances of all child types is limited by the *multiplicity* set for the parent type. Similiar for edges inherited from a parent node, the overall number of incoming or outgoing edges is limited by the *multiplicity* set for the edge connected to the parent type. Now it is possible to define *abstract* rules, containing ancestor nodes.

1.5 Drawing and Modifying Nodes and Edges

Having defined node and edge types, the user changes to the *Draw* mode by selecting *Draw* in the menu Mode. Now nodes may be drawn by clicking the left button on the panel of the graph or rule editor. An edge between two nodes is created by clicking on the source first and then on the target node of the edge using the left mouse button. Edges are all directed. An edge may contain one bend. There are two possibilities to draw a bend: Firstly, the bend can be inserted when creating the edge by clicking on a place in the panel between clicking on source and target nodes.
Secondly, the bend can be created or moved being in the Move mode of the editor by grabbing the edge with the left mouse button at the center point and dragging it to the desired place in the panel. Bend points can be removed by going to the center point of an edge and selecting item Straighten in its context menu. Bends can also be removed by selecting one or more edges with bends and choosing item Straighten in the Edit menu.
Being in the Select mode, a graph object can be selected by clicking on the object. A selected object is colored in green. Selecting an object twice means deselecting it. Selected objects may be duplicated, moved, deleted, mapped or unmapped.
Being in the Move mode, a graph object can be moved by pressing the left mouse button when the cursor points to a node or an edge bend, and dragging the mouse and release it. When a node is moved, incident edges are moved accordingly. The nodes and bends can also be moved by pressing and dragging the middle mouse button. In this case, the switch to the Move mode is not necessary. Moving leads to the parallel translation of the selected objects. If an object is moved outside the current editor panel, the size of the panel will be enlarged. The scroll bars then allow to access the invisible object. The nodes or bends can also be moved by middle mouse button.

An alternative to change between edit modes is to use the context menu Edit Mode that pops up by clicking the right mouse button when the cursor placed on the background of the editor panel. Here, the user can switch between general edit modes (like Draw, Select, Move, Attributes, Map, Unmap, Image_view). The selected mode is only valid for a local editor panel.

Here is a short description of items of menu Mode:

- Draw Mode for drawing nodes and edges.
- Select Mode for selecting nodes and edges.
- Move Mode for moving nodes and bends of the edges.
- Attributes Mode for setting attributes of the nodes and edges.
- Map Mode for setting rule or match morphism mappings of the nodes and edges.
- Unmap Mode for unsetting rule or match morphism mappings of the nodes and edges.
- Image_view If this mode is set, the icons will be used to visualize the nodes of all graphs. Such icons may be set only for a node type using item Image of the pop-up menu of Node Type. A Load File dialog helps you to set the icon files. NOTE: The location directory of these icon files has to be added to the CLASSPATH environment variable.

Another context menu containing graph object specific operations is evoked by a click on the right mouse button if the cursor is over a node or the small black rectangle of an edge. The operations here (like evoking the attribute editor, Delete, Copy, Select, Map, Unmap) are valid only for the graph object the menu had been opened for. A third operations menu pops up when the current graph object is selected before evoking the menu. Figure 3 shows different context menus.

Figure 3: Context Menus for Editing Operations

Please note that some items of the pop-up menu Operations are disabled. This depends on the context of graph objects:

- If a graph object is a node, item Straighten is disabled.
- If a graph object belongs to the type graph of a grammar, item Multiplicity is enabled, otherwise - disabled.
- If a graph object belongs to the left graph of a rule, item Map is enabled, otherwise - disabled. You may use it to define a mapping of a rule, NAC or match morphism. The current graph object will be the source of mapping. Now, clicking on another graph object to defines the target of the mapping.
- Item Unmap is always enabled. It can be used to delete already defined mapping of a rule, a NAC or a match morphism.

1.6 Editing Rules

A rule consists of a left hand side (LHS), a right hand side (RHS) and a morphism from LHS to RHS which maps part of the LHS to RHS. Editing a rule, it is advisable to draw the objects of the left-hand rule side first, and then choose menu item Identic Rule in menu Edit. The graph objects of the left-hand rule side are then copied to the right-hand rule side, and an identic rule morphism is generated. The rule morphism is indicated in the editor by showing identic numbers for each graph object on the left-hand side and its image on the right-hand side of the rule. Now, having two identic rule sides, the graph objects that are to be deleted by the rule, have to be deleted in the right-hand rule side, and graph objects that are to be created by the rule have to be drawn there additionally. Another possibility to draw the same rule would be to draw the part of the left-hand rule side first that is to be preserved by the rule, then choose Identic Rule and afterwards add those graph objects to the left-hand rule side that are to be deleted and those graph objects to the right-hand rule side that are to be created by the rule.
An alternative to define a rule morphism is to draw both rule sides first, and then change to the edit mode Map. In this mode, the rule morphism can be defined elementwise by clicking on a graph object in the left-hand rule side first, and then clicking on a graph object of the same type in the right-hand rule side. This method is useful when there is already a rule morphism and the user wants to change it. One can define an injective or non-injective rule morphism. A non-injective mapping of objects is indicated by several numbers in the target object to which several source objects are mapped to.
To delete the mappings of a rule morphism the edit mode Unmap can be used and then click on a certain graph object.
Analogously, a NAC may be entered by drawing graph objects into the NAC graphical editor and then defining the mappings from the left-hand rule side to the NAC by the Edit Mode Map clicking elementwise on a graph object in the left-hand rule side first, and then clicking on a graph object of the same type in the NAC graph. In addition to mapping graph objects of the same type it is possible to map a LHS object to a more refined type

in the NAC graph, if there is a parent child relation between those types. Using menu item Identic NAC of menu Edit functions similar to menu item Identic Rule.

1.7 Adding Attributes to a Graph Object

Attributes of graph objects have three components: the attribute type, its name and its value. Taking Java objects as attributes, the corresponding Java class defines the attribute type. The object itself is the attribute value.
Attribute values in a host graph have to be constant values (like integer numbers for example). The right-hand side of a graph rule can contain Java expressions which are instantiated by Java objects, at runtime.
The left-hand side or a NAC of a rule may contain constants or variables as attribute values, but no expressions like for example `a = x + 1`. A NAC may contain the variables declared on the left-hand side or a new variable declared as an input parameter. Variables are usually implicitly declared by just using them as attribute values. In this way they also get a type, i.e. the type of the current attribute instance. The scope of a variable is its rule, i.e. each variable is rule-globally defined. In a rule's right-hand side, Java expressions are allowed. They may contain the variables declared on the left-hand side or a new variable declared as an input parameter. Multiple usage of the same variable is allowed and required different attribute values to be equal.

1.7.1 Attribute Editor

In AGG the attribution of objects is performed in a special editor. There are different ways to invoke the attribute editor. A convenient one is to click on a node or edge with the right mouse button and choose the item Attributes .... Another possibility to invoke the attribute editor is to select a node or edge and choose the Edit menu item Attributes. Please note that only one graph object can be edited at once. So there is an error message if more than one object is selected. Last but not least, there is the possibility to invoke the attribute editor by switching to the mode Attributes ... in the Mode menu and then clicking on the object which attributes are to be edited. The mode Attributes is also available in the pop-up menu Edit Mode as shown above.

There are several occurrences of attributes: in type graphs, rules, host graphs, graph constraints. If attributes are edited in a rule, the attribute editor is shown instead of the host graph. To get the host graph back, click on the graph in the GraGra tree.
The attribute editor allows to declare a *type*, a *name* and a *value* for an attribute. Additionally, the visibility of an attribute can be changed between shown and hidden. There may be multiple attributes for one type of graph object. As the underlying attribute handler interprets expressions in Java syntax, the types for attributes are either primitive Java types (i.e. byte, short, char, int, double, long, float, boolean) or Java classes. Java expressions are arithmetic expressions (e.g. `x + 1`), boolean expressions (e.g. `width <= 7`), bit manipulations, conditional expressions (e.g. `v>8?7:0`), operations on arrays, creating a new object instance by calling a constructor of a class (e.g. `new Vector()`),

access to class and instance variables and invocation of class and instance methods (e.g. `java.lang.Math.sin(0.5)` or `v.elementAt(i)`). Please note, if a method returns no object (is void), we will use ;(semicolon) to return the object (e.g. `v;addElement(new Integer(i))`). Apart from the standard Java library JDK[1], user-defined classes can also be used for attribution. The CLASSPATH environment variable and the class libraries have to be adapted accordingly. Alternatively, a configuration editor can be used.
As an example for a user-defined Java class we will use a class `Entry` of our *HTML_Browser* example. This class defines the operations for getting directories and files and displaying a file in a browser. It will be shown in Section 1.8.
Figure 4 shows the attribute editor for attribute declaration and value definition. The attribute editor has been invoked for the black unnamed edge in the left-hand side of the rule *expand* (Fig. 15) of the grammar shown in the next section. This edge type has one attribute of type *int* with name *d* set to variable x.

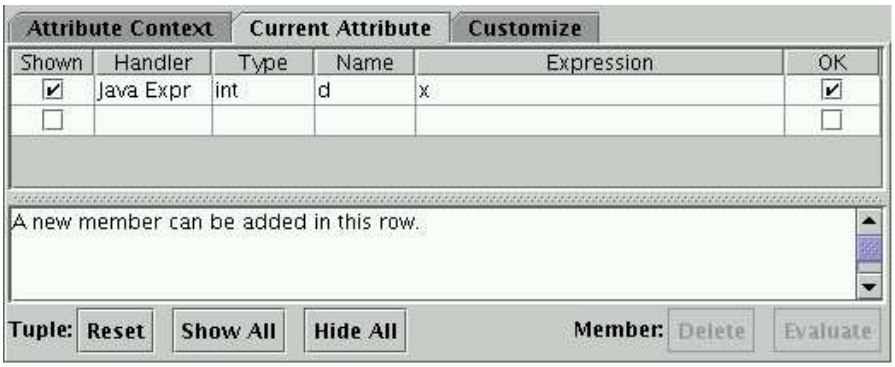


Figure 4: The Attribute Editor

Figure 5 shows the attribute context editor for variable declaration and attribute condition definition of the rule *expand*.

Let us shortly explain the other window elements of the attribute editor. First of all the attribute editor is devided in three editors which can be selected by clicking of the according tab at the top of the attribute editor.
The three editors are called according to the appearance in the attribute editor from the left to the right: attribute context editor, current attribute editor and customize editor.
If you invoke the attribute editor, the current attribute editor is shown (see Figure 4). The current attribute editor is devided into three parts. At the top is a table which holds all attributes. One attribute is called member and all together are a tuple. The attribute editor knows two orders of members. The first order is the one how the user inserted the members. The second one is how they are displayed. That means a member can be at the last position which was inserted at first. To change the display order of the attributes use the middle mouse button.
In the middle, there is a text area which displays help messages and error reports. At

[1] Java Development Kit

the bottom, a button bar occurs which get you access operations on attributes. The Reset button sets all layout changes to the initial state. The Show All button marks all members visible at once. The Hide All button does it vice versa. The Delete button deletes the marked member. NOTE: This takes affect on the tuple type and all its instances. The Evaluate button tries to evaluate the expression in the marked line.
Whenever the attribute editor is opened for another graph object of the same type, the names and types of all attribute members edited for this type before are shown in the list. Only the values have to be entered for each new graph object.

1.7.2 Attribute Context Editor

The attribute context editor contains two editors (see Figure 5). The left one contains the context's variables and the right its application conditions. Each one has a little text area for messages like the current attribute editor has. Variables as well as conditions can be moved up and down in there tuples but cannot be made invisible. The variable editor additionally has two more columns prepended. IN is a switch for the variables parameter property. If it is checked the variable in that line is an input parameter. OUT switch is not jet supported. At the bottom, the buttons of these editors are a subset of those occurring the current attribute editor. Figure 5 shows those variable declarations and attribute conditions defined for rule *expand* in Figure 15).

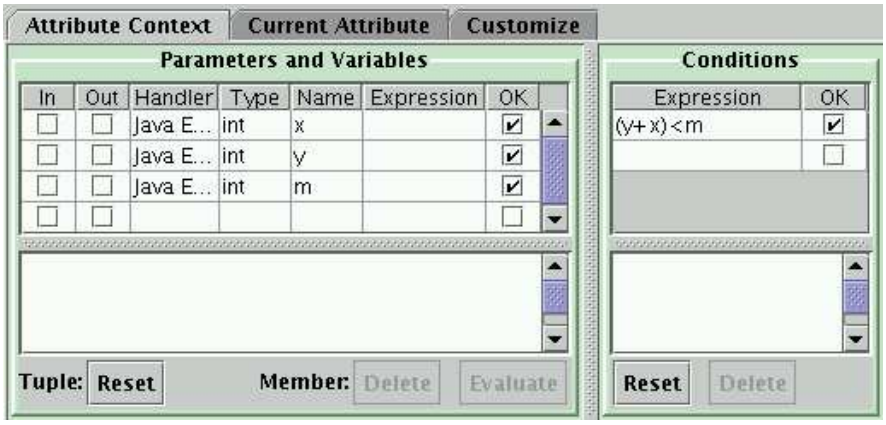


Figure 5: The The Attribute Context Editor

1.7.3 Attribute Customize Editor

The customize editor contains two more editors. Figure 6 shows the manager editor. It displays the registered attribute handlers. Uptonow, there is only one available which handles Java expressions.
The handler editor consists of packages which are defined to search for classes. This can be compared to the set of import statements in Java programs. This part is very important if you want to use a class which is embedded in a non-standard package. E.g. you want

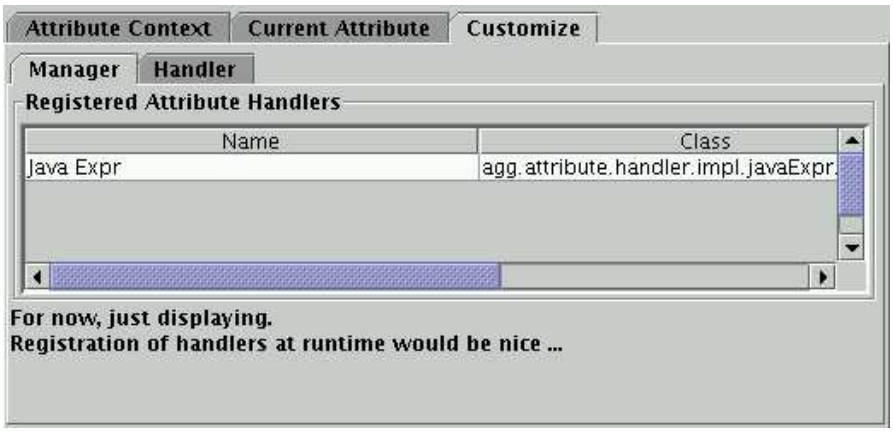


Figure 6: The Customize Manager Editor

to use class *Pair* from Objectspace. The CLASSPATH environment variable has to be set to the package which contains *Pair*. That means the JAR file. Thereafter in the handler editor, the package has to be inserted. In our example *com.objectspace.jgl* must be inserted. Figure 7 shows the handler editor.

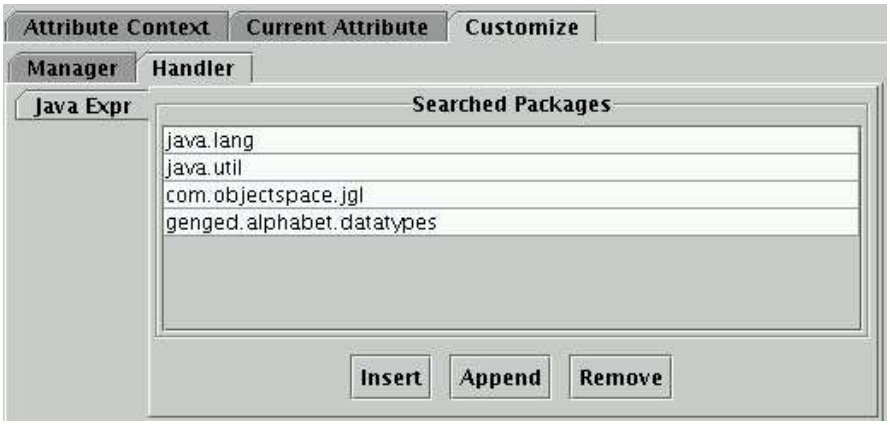


Figure 7: The Customize Handler Editor

1.8 An example for a user-defined Java class

The HTML Browser grammar is an example for visualizing a directory tree and browsing HTML files. This grammar performs a recursive scan of a directory of your filesystem for HTML files and represents the directory tree as a graph. Afterwards, it starts up an HTML browser to inspect the files that have been found. The files already visited are marked in the graph. This example demonstrates how you can access system resources like the file system using user-defined Java classes and their methods in graph rules. It also shows how you can control an external Java application (here: the browser) by graph

transformation rules.
Figure 8 shows rule *initDir* being applied to the start graph of grammar *HTML_Browser*. The rule launchs a dialog asking for the directory to browse.

Figure 8: The *initDir* rule

Here the user-defined Java class `Entry` (`Entry.java`) is used for attribute *name* in the right-hand side of the rule.
This class defines operations for:

- getting directory and file: methods `getDirEntry` and `getFileEntry`,
- getting their numbers: methods `getNumberOfDirs` and `getNumberOfFiles`.

Also rules *showDir*, *prepare2ndPath* and *showFile* invoke methods of this class.
Rule *openBrowser* in Figure 9 uses another user-defined Java class `Browser` (`Browser.java`) to create a file browser.

Figure 9: The *openBrowser* rule

Finally, rule *viewPage* in Figure 10 uses class `Browser` to view a HTML file. Please pay attention to the attribution of `ref` in the *Browser* node in the RHS of the rule. The method `setPage` of class `Browser` returns no object (is void) but we want to return. In

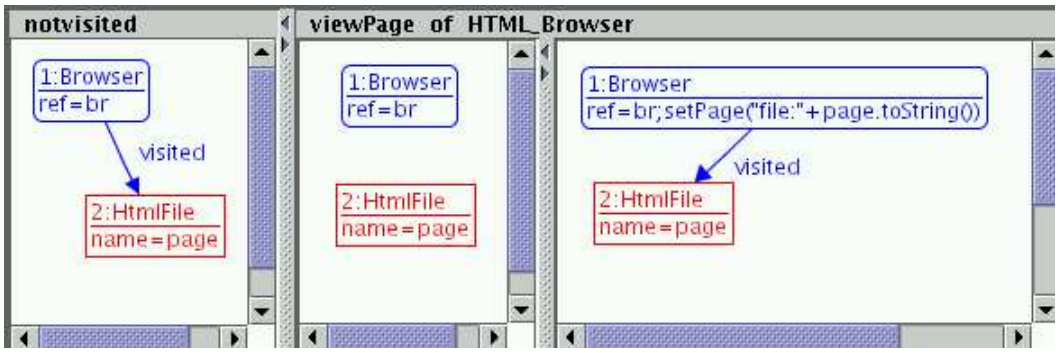


Figure 10: The *viewPage* rule

this case we use ; (semicolon) to return the changed Browser object ($br; setPage("file:" + page.toString())$).
Please note: A user-defined class has to contain the default class constructor without parameters. If a class rewrites the default class constructor, the first line has to contain *super();* entry. Moreover, *setter* and *getter* methods have to be defined for class parameters like this: for a parameter with the name abc the *setter* and *getter* methods will get the names *setAbc(...)* and *getAbc()*. The method *toString()* can be rewritten to get a sufficient string representation of this class.

1.9 Saving and Loading, Exporting and Importing Graph Grammars

AGG supports not only the possibility to open and save graph grammars in XML (`http://www.w3.org/TR/REC-xml`) format using common DTD (`http://www.w3.org/TR/REC-xml\#dt-doctype`) , but also exports and imports format GXL (`http://www.gupro.de/GXL`) and exports GTXL format (`http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html`). (GXL is an XML-based file exchange format for graphs and GTXL is an XML-based file exchange format for graph transformation systems.)

AGG uses the XML format GGX as external data format.
Menu File contains the following menu items:

- New GraGra - Create a new graph grammar.
- Open - A file open dialog helps you to search through file directories.
- Save - Save the current grammar in the current file. If the current grammar is new, a file save dialog will appear for saving.
- Save As - A file save dialog helps to search through directories. The file name will get extension `.ggx`.
- Open (Base) - Similar to menu item Open. But the graph layout information (such as (X,Y) position of the nodes) will get lost. The default graph layout algorithm of AGG will be used.

- Save As (Base) - Similar to menu item Save As. But only basis graph structure of the grammar will be saved. The graph layout information will get lost.
- Export - Exports the current grammar to
  - GXL format using ggx2gxl.xsl (`http://tfs.cs.tu-berlin.de/agg/xml/ggx2gxl.xsl`) and gxl.dtd (`http://tfs.cs.tu-berlin.de/agg/xml/gxl.dtd`).
  - GTXL format using gts2gtxl.xsl (`http://tfs.cs.tu-berlin.de/agg/xml/gts2gtxl.xsl`) and gtxl.dtd (`http://tfs.cs.tu-berlin.de/agg/xml/gtxl.dtd`).

  A file save dialog helps you to search the file directories. In case of GXL the file name will get extension `.gxl` and in case of GTXL it will get `.gtxl`.
- Import - Imports an external graph from
  - GGX format to integrate an external graph into current grammar.
  - GXL format using gxl2ggx.xsl (`http://tfs.cs.tu-berlin.de/agg/xml/gxl2ggx.xsl`) and gts.dtd (`http://tfs.cs.tu-berlin.de/agg/xml/gts.dtd`) and agglayout.dtd (`http://tfs.cs.tu-berlin.de/agg/xml/agglayout.dtd`).
  - OMONDO XMI format using omondoxmi2gxl.xsl (`http://tfs.cs.tu-berlin.de/agg/xml/gxl2ggx.xsl`) and gts.dtd (`http://tfs.cs.tu-berlin.de/agg/xml/gts.dtd`) and agglayout.dtd (`http://tfs.cs.tu-berlin.de/agg/xml/agglayout.dtd`).

  An external graph can be integrated into the current graph grammar in place of the host graph, if the type set of a grammar contains all types of this external graph. Otherwise, the external graph will be integrated into a new generated grammar with an empty rule set and a type set similar to the type set of the external graph.
  A file load dialog helps you to search through file directories.
- Close GraGra - Closes current graph grammar. If the grammar has been changed, you will get a warning to save it.
- Quit - Exits from AGG application.

Additionally, Save, Save As and Close are also available by pop-up menu GraGra.
And finally, menu item Reload of pop-up menu GraGra allows reloading the current graph grammar.

1.10 Layering

AGG supports the possibility to set rule layers and thus to define layered graph grammar. The layers fix the order of how rules are applied. The interpretation process first has to apply all rules of layer 0 as long as possible, then all rules of layer 1, etc. Rule layers allow to specify a *simple control flow* graph transformation. Once the highest layer has been

finished the transformation stops, unless option loop over layers is checked.

Moreover, rule layers allow some kind of optimization :

- During the critical pair analysis, critical pairs are searched for rules in the same layer only.
- During parsing, first all rules of one layer are applied as long as possible, before the rules of the next layer are applied.
- During termination checking, the termination criteria are based on assigning a layer to each rule, node and edge type.

The graphical user interface for setting rule layers is available by item Set Rule Layer in pop-up menu GraGra. The dialog is shown in Figure 11. Another possibility to set rule layers is to use item Set Layer in pop-up menu Rule.

Figure 11: Rule layer dialog

Using item Sort by Rule Layer in pop-up menu GraGra the rules of a grammar are sorted by layer.

1.10.1 Layers with Trigger Rule

In this AGG version for each layer one rule may be determined as trigger for this layer. The so-called trigger rule is applied as first and only once, all other rules are applied as long as possible.
If the trigger rule is not applicable, the corresponding layer is not executed at all.
A trigger rule can be set using item Set Trigger Rule for Layer of the GraGra pop-up menu. A table dialog *Set Trigger Rule for Layer* appears. Each layer may have at most one *trigger rule* that is set using check box Trigger.
In the gragra tree view, *trigger rules* are collored in red.
Trigger rule settings will be saved and loaded with the layered graph grammar.

Please note: In the table dialog *Set Trigger Rule for Layer* it is allowed to change layers of rules, too. Selecting or changing layers can effect also the trigger selections. So after changing a rule layer, the user needs to check the trigger definition and possibly reset it again.

2 Rule Application

We now explain the definition of match morphisms and the different modes for rule application to the host graph in the context of example *ShortestPath*, computing the shortest path between two nodes of a given graph.

2.1 The Step Mode

The first mode to apply a rule is called *Step* mode. Here, one selected rule will be applied exactly once to the current host graph. The match morphism in the *Step* mode may be defined by the user analogously to a rule morphism: From menu Transform, we choose menu item Match and define the match morphism elementwise by clicking on a graph object in the left-hand rule side first, and then clicking on a graph object in the host graph. Of course, the corresponding graph objects must be of the same type or in case of inheritance connected through a parent-child relation, there must be a consistent mapping of the attributes, and the mapping must be compatible with the source and target functions of edges. Another possibility to define a mapping is to choose the Map mode of the context menu Edit Mode and then click on the corresponding objects in the LHS and in the host graph. Of course, a mapping also can be deleted by choosing Unmap from the same menu. Defining the match completely "by hand" may be tedious work. Therefore, AGG allows to complete a partial match by choosing the Transform menu item NextCompletion after having defined an arbitrary partial morphism. The partial morphism will be completed to a total match automatically. If there are several choices for completion, one of them is chosen arbitrarily. Calling Next Completion again, another completion is computed, if possible. In this way, all possible completions are computed and shown one after the other in the editor using equal numbers for corresponding objects. After having defined the match, we choose item Step in menu Transform, and the rule will be applied to the host graph once at the given match. The result is shown in the graph editor. Thereafter, the host graph can immediately be edited, for instance to improve the layout of the new graph. Afterwards, any other rule may be selected and applied to the new graph in the *Step* mode as described above.

In AGG transformation options can be chosen by clicking item Options... of menu Preferences. The options are shown in Figure 12.

The first option defines a match completion strategy:

- CSP - Constraint Solving Problem,
- CSP w/o BJ - Constraint Solving Problem without Back Jumping,

- Simple BT - Simple Back Tracking.

More details described in the paper on efficient graph pattern matching by Michael Rudolf (`http://tfs.cs.tu-berlin.de/agg/match.ps`).

Figure 12: Transformation Options

For a very simple host graph (for example without attributes), the simple strategy Simple BT might be the fastest, whereas for our ShortestPath example we chose CSP which can be considered as standard strategy .
Options injective, dangling, identification and NACs define the properties of selected match

completion strategy for direct transformation step.
Since AGG has a formal foundation based on the algebraic approach to graph transformation and the theoretical concepts are implemented as directly as possible AGG offers clear concepts and sound behavior concerning the graph transformation. Generally, to define a direct transformation step we have used the so-called single-pushout (SPO) approach to graph transformation in the category of graphs and partial graph morphisms. Switching on the gluing condition, combining the dangling and identification conditions, allows to realize the so-called double-pushout (DPO) approach in the category of graphs and total graph morphisms in the operational point of view.

Furtheron, two options allow to decide how graph consistency constraints influence the graph transformation process.

- Option consistent transformation only means that graph transformation steps with a consistent result graph will be performed only. The graph transformation process will not be stopped.
- Option stop after inconsistent transformation means that the graph transformation process will be stopped after the first transformation with an inconsistent result graph.

These options have an effect only, if the graph grammar contains graph consistency conditions.

Options for layered rule application have the following meaning:

- Option layered allows to use rule layers during graph transformations. Generally, rules given by a graph grammar are applied *non-deterministically*. A rule layer can be set for each rule or a set of rules. These layers fix the order how rules are applied. The interpretation process first has to apply rules of layer 0 as long as possible, then rules of layer 1, etc. That gives the possibility to specify a simple control flow on graph transformation.
- Option show layer before transform gives the possibility to change the layers before the graph transformation starts.
- Option loop over layers allows to continue graph transformation by beginning on the smallest layer again and continuing as long as possible.
- Option stop current layer only means that the graph transformation on the current layer will be stopped, if menu item Stop of menu Transformation is chosen. Thereafter, rules of the next layer will be applied, etc.

If option check rule applicability on the host graph is set, each rule will be checked whether it is applicable to the current host graph. This check will be done after each graph transformation step. Visually, the name of a non-applicable rule will be gray-colored in the

grammar tree view. Alternatively, you can use menu item Check Rule Applicability of pop-up menu GraGra to perform this check only once. Using button Undo allows to switch off this check.

Further options define a kind of display settings of the graph transformation process.

- Option show after step causes that the altered host graph will be shown after each graph transformation step during the interpretation process. If this option is not set, the altered host graph will be shown at the end of the transformation process only.
- Option wait after step means that the transformation process will be paused after each succesful match and step to give the possibility to see the current match and a rule of the application. Entering any key allows to continue the transformation process.
- Option select new objects after step means that the new graph objects will be selected after each step to improve the visualization of the graph transformation process.

To get the previous state of the host graph after a transformation step or process has been done you can use menu item Undo of menu Transform. Alternatively, you can use menu item Reset Graph of pop-up menu GraGra.

Let us illustrate the transformation process by transforming the start graph of our *ShortestPath* example using the rules of graph grammar *ShortestPath.ggx*. We will go through a transformation session in *Debug* mode in order to show all important intermediate transformation steps.

2.1.1 A Sample Run

In Figure 1 the start graph of grammar *ShortestPath.ggx* is depicted, consisting of some cities connected by roads.
The rules of the grammar are shown in Figures 13 - 21.

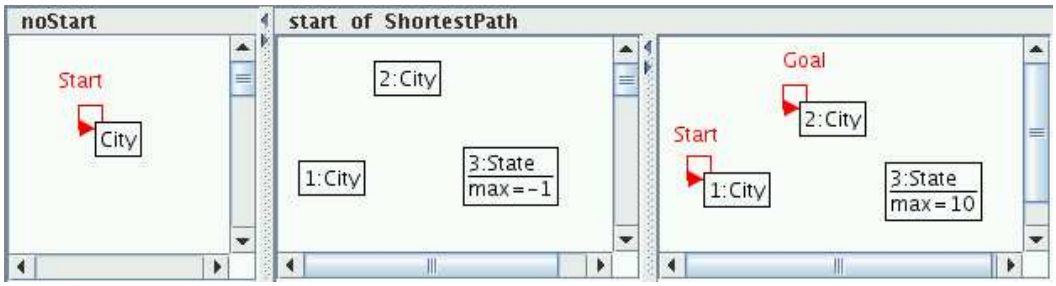


Figure 13: Rule *start*

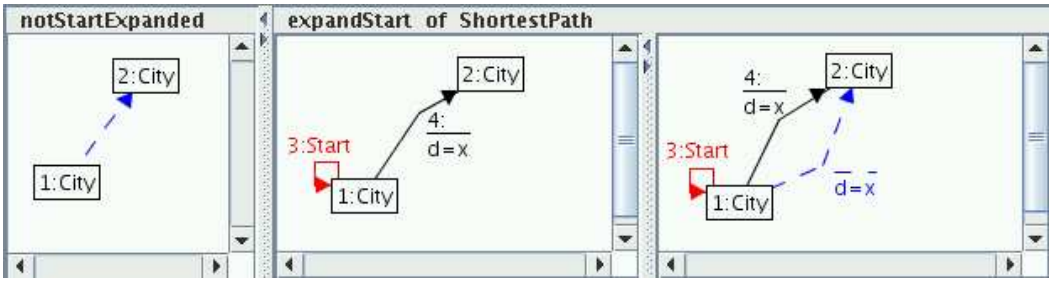


Figure 14: Rule *expandStart*

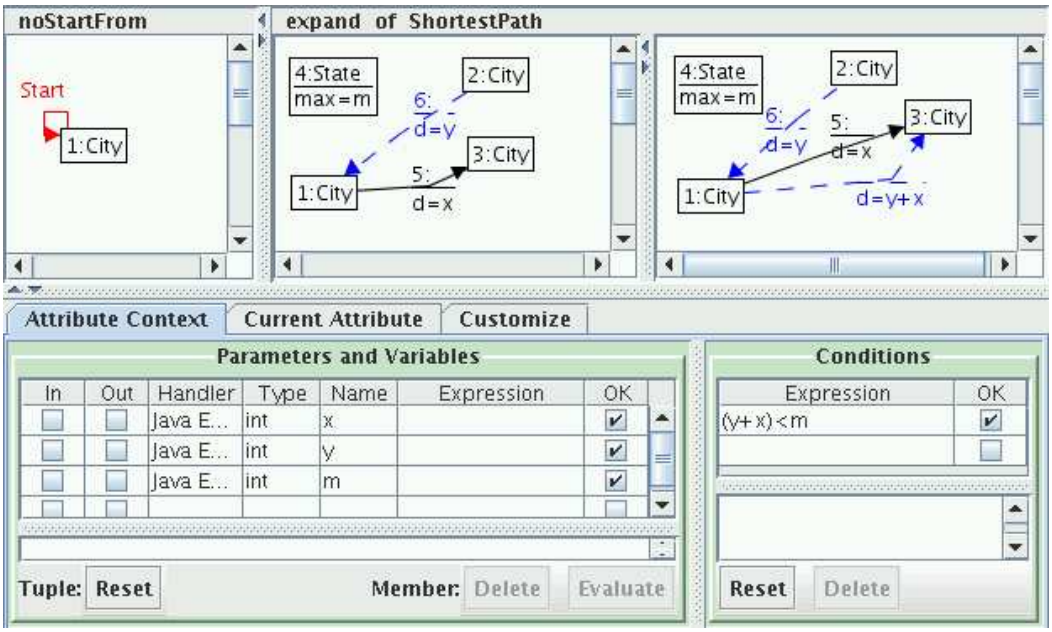


Figure 15: Rule *expand*

Figure 16: Further NACs of Rule *expand*

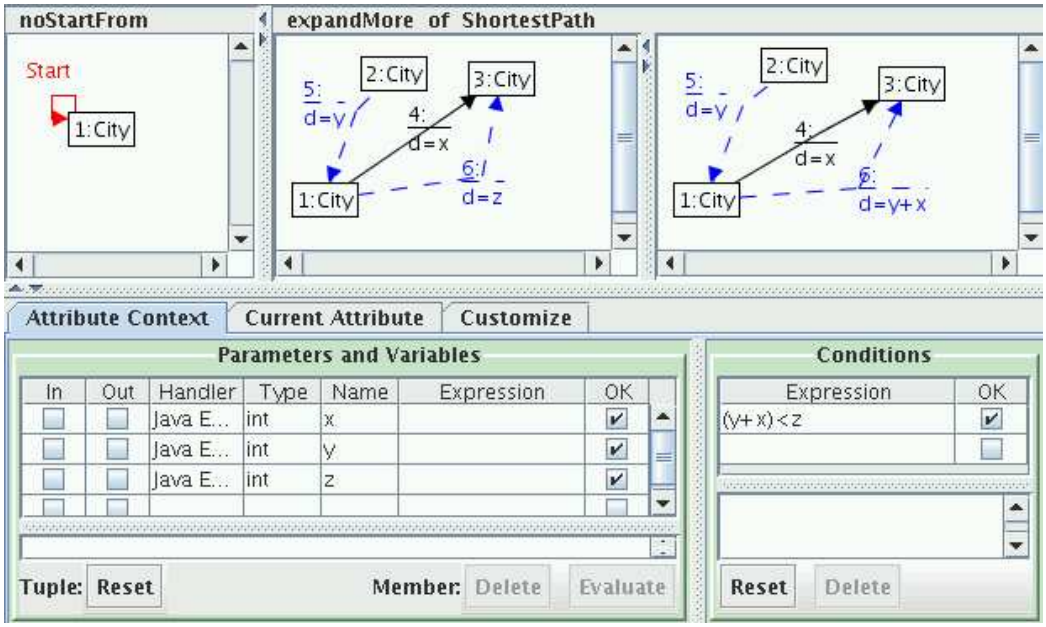


Figure 17: Rule *expandMore*

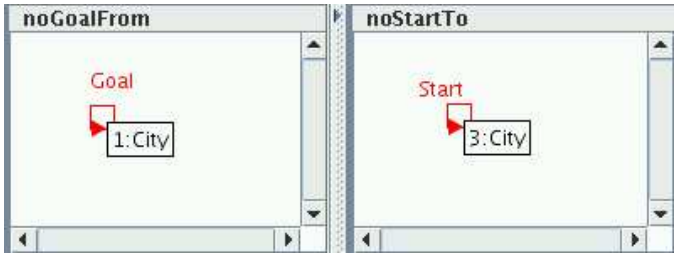


Figure 18: Further NACs of Rule *expandMore*

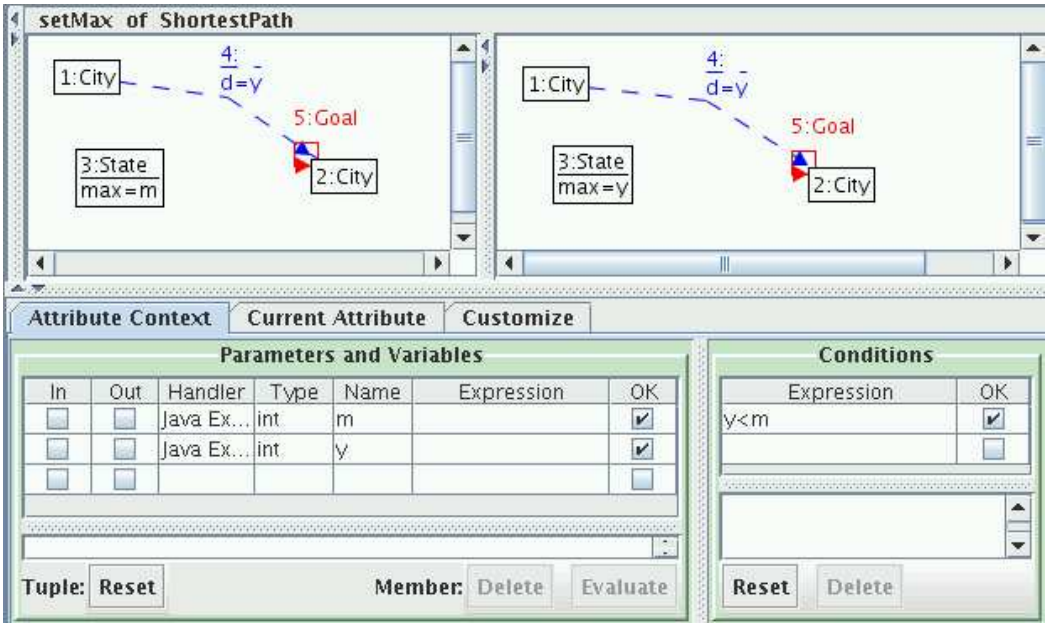


Figure 19: Rule *setMax*

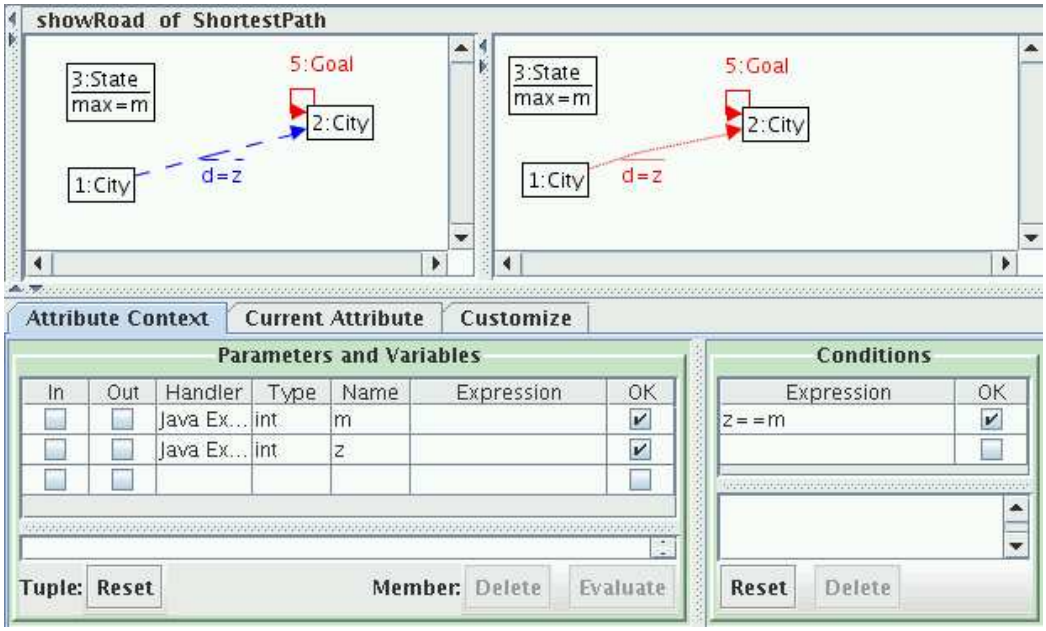


Figure 20: Rule *showRoad*

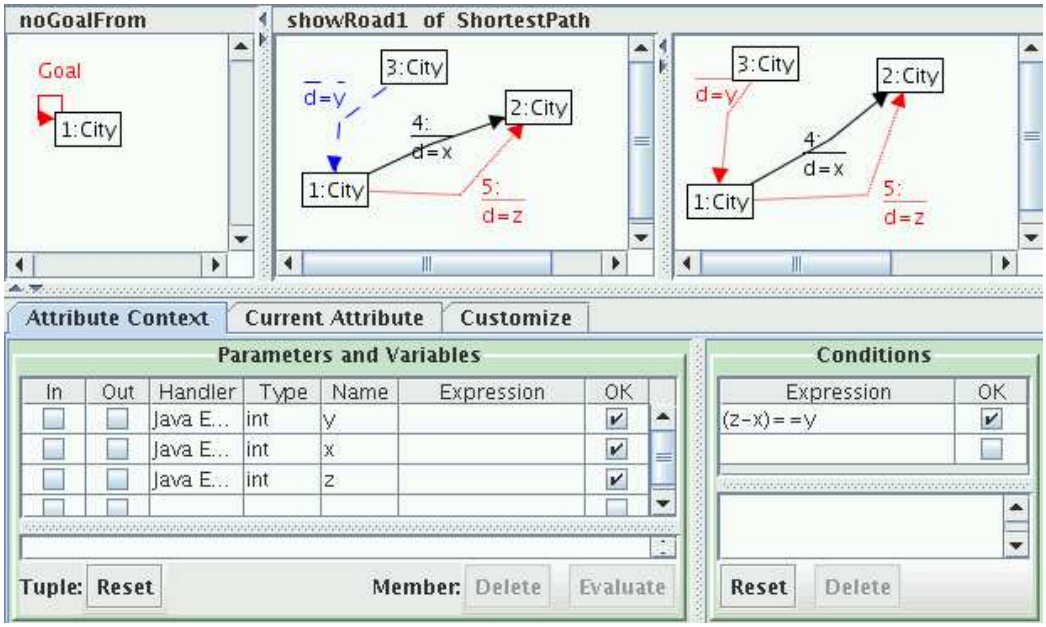


Figure 21: Rule *showRoad1*

Figure 22: Further NACs of Rule *showRoad1*

Transformation options chosen for our example are injective matches which satisfy the dangling condition, rules with NACs, the completion strategy *CSP* and layered rules as shown in Figure 23.

Figure 23: Rule layers of grammar *ShortestPath*

To determine a start and a goal city, we have to define a partial match of rule *start* by mapping the two cities at the left-hand side. After completing this partial match and applying the rule, the transformation result is shown in Figure 24.

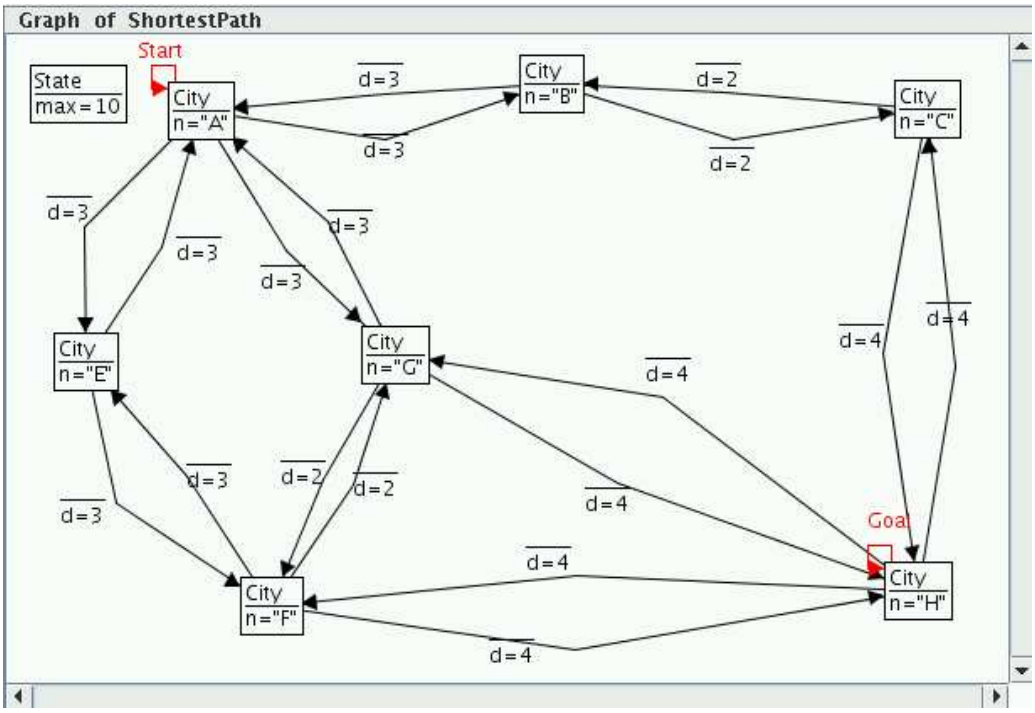


Figure 24: The host graph after applying *start* rule (in Fig. 13)

The match defines city A to be the start and city H to be the goal of the search. Additionally, rule *start* defines a maximal distance between start and goal cities. This is done by a node $State : Exp$ where attribute max is set to 10.

Figure 25 shows the host graph after applying rule *expandStart* three times. Three possible matches of this rule realize the expansion of the start city: for all three neighbours the corresponding edges are generated. The newly generated edges carry the start information of the paths with length from City A to City H.

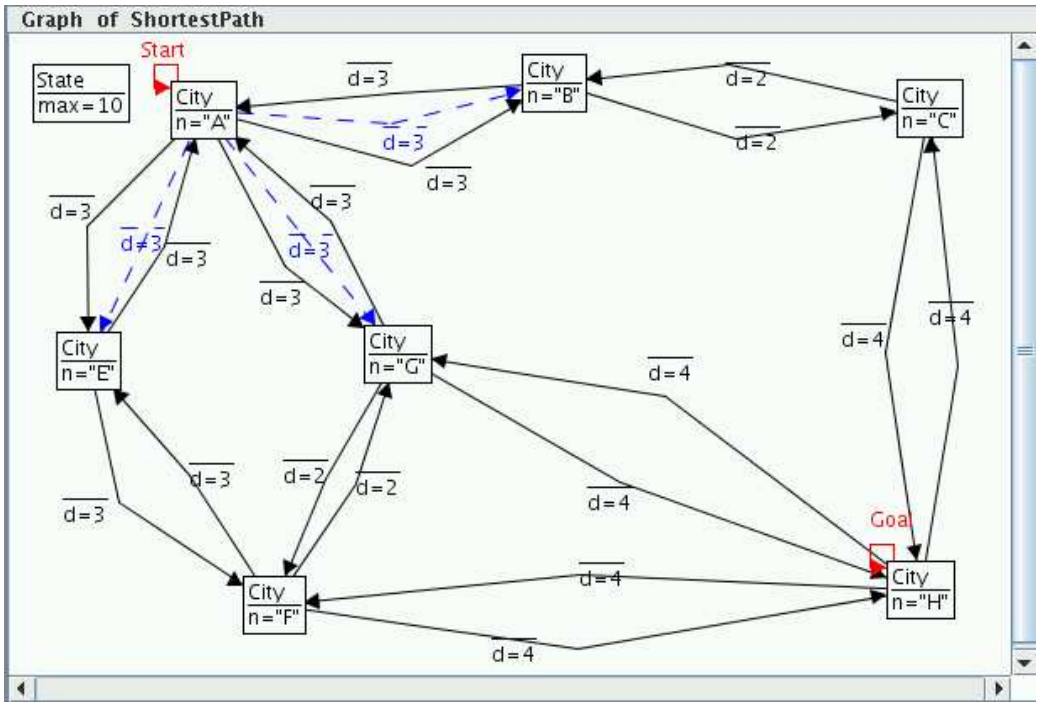


Figure 25: The host graph after applying *expandStart* rule (in Fig. 14) three times

In Figure 26, we see the host graph after applying rule *expand* nine times. New edges are generated to all possible cities on the way to the goal. The path lengths have been raised. This rule has an attribute condition $(y + x) < m$ that forbids to follow a path with a new length longer then max m. Four NACs of this rule prevent: 1. the current city is a start city, 2. the current city is a goal city, 3. the next city is a start city, 4. the path to the next city is already expanded.

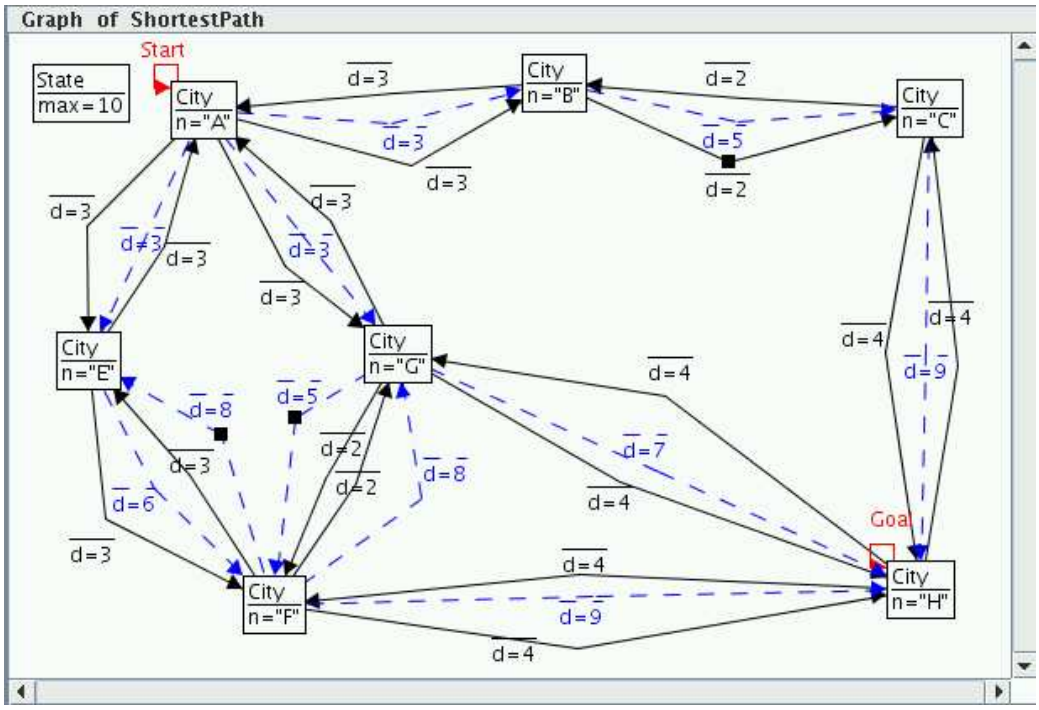


Figure 26: The host graph after applying *expand* rule (in Fig. 15) eight times

The next rule to apply is *expandMore*. That rule compares already computed path lengths and takes the smallest as a best possible path. The NACs of the rule prevent: 1. the current city is a start city, 2. the current city is a goal city, 3. the next city is a start city. In our example this rule is not applicable for chosen the *Start* - *Goal* pair of cities.

Now the rule *setMax* is applicable. In Figure 27, we can see the host graph where attribute *max* of node *State* is changed from 10 first to 9 and then to 7.

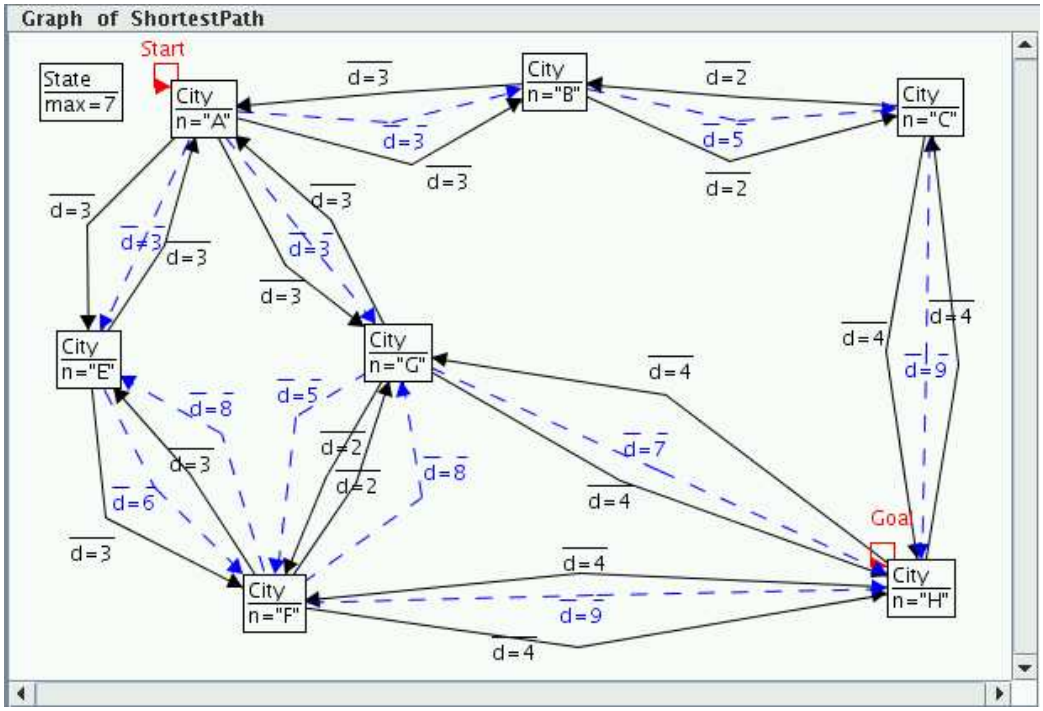


Figure 27: The host graph after applying *setMax* rule (in Fig. 19) two times

Finally, Figure 28 shows the host graph after applying *showRoad* and *showRoad1*. Rule *showRoad* marks the last part of the path from start to goal cities and rule *showRoad1* marks the next part of the path backwards.

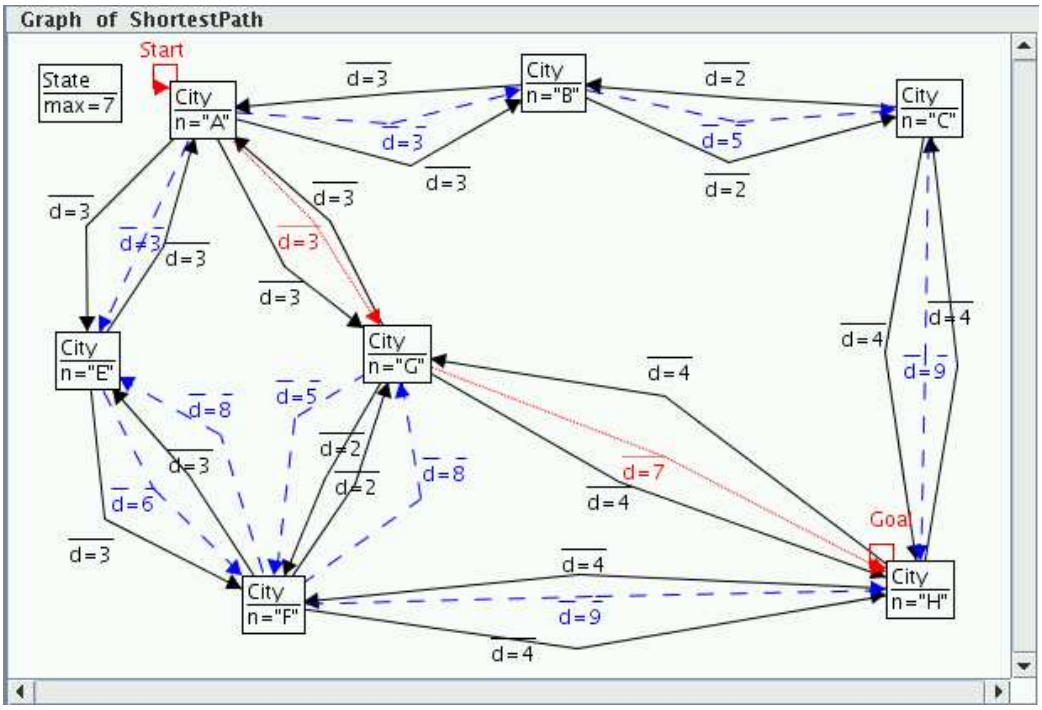


Figure 28: The host graph after *showRoad* (in Fig. 20) and *showRoad1* (in Fig. 21) applied once each

Figure 28 shows a successful result of the execution of our grammar.
You can see the shortest path from *Start* to *Goal* city with length $7 < 10$: A - G - H with *State* max=7.

2.2 The Interpretation Mode

The second mode to realize graph transformations is called *Interpretation* mode. This is a more sophisticated mode, applying not only one rule at a time but a whole sequence of rules.
The order of rules to be applied is defined non-deterministically in general. Starting the interpretation by clicking on the Start button in menu Transform, a rule is chosen randomly and applied. Thereafter, the next rule is chosen randomly and applied, and soon.
Using rule layers, as in the example above, we can define some kind of control flow on graph transformation. Such transformations are called layered graph transformations.
The graph transformation will stop if either there are no more rules applicable, or the user has chosen item Stop in menu Transform.
In this mode transformation options show after step, wait after step and select new objects after step can help to analyse the interpretation process.

3 Analyzing Graphs and Graph Grammars

AGG has a formal foundation based on the algebraic approach to graph transformation. Due to its formal foundation, AGG offers the following validation support:

- consistency checking of graphs and graph transformation systems,
- conflict, dependency detection of graph transformation rules using critical pair analysis,
- graph parsing,
- termination criteria for layered graph grammars.

3.1 Consistency of Graph Grammars

AGG provides the possibility to formulate consistency conditions which can be tested on single graphs, but which can also be shown for a whole graph transformation system.
Herewith we have in AGG a consistency control mechanism which is able to check certain *consistency conditions* for a given graph.
Generally, *consistency conditions* (CC) describe basic (global) properties of graphs as e.g. the existence of certain elements, independent of a particular rule. But we also provide a transformation of global consistency conditions into *post application conditions* for individual rules. A so-constructed rule is applicable to a consistent graph if and only if the derived graph is consistent, too. A graph grammar is consistent, if the start graph satisfies consistency conditions and the rules preserve this property.
If a consistency condition holds for a graph transformations system, all derived graphs satisfy this condition.

Consistency of conditional graph grammars is described in:

- "Consistency of Conditional Graph Grammars - A Constructive Approach" by Reiko Heckel and Annika Wagner, 1995 (`http://www.elsevier.nl/locate/entcs/volume2.html`).
- "Modellierung und Nachweis der Konsistenz von verteilten Transaktionsmodellen für Datenbanksysteme mit algebraischen Graphgrammatiken" by Manuel Koch (in German) (Bericht-Nr. 96-36. Forschungsberichte des Fachbereichs Informatik an der Technischen Universität Berlin).
- "Konzeption und Implementierung eines Verfahrens zum Nachweis der Konsistenz in einer attributierten Graphgrammatik" Diploma thesis of Michael Matz (in German) (`http://tfs.cs.tu-berlin.de/agg/consistency/MMatzDiplomarbeit.ps`).

3.1.1 Graphical Consistency Constraints

In AGG *graphical consistency constraints* can be defined.
A *graphical consistency constraint* is a total injective morphism $c : P \rightarrow C$, the left graph P is called premise and the right graph C is called conclusion. A *graphical consistency constraint* is satisfied by a graph G, if for all total injective morphisms $p : P \rightarrow G$ there is a total injective morphisms $q : C \rightarrow G$ such that $q \circ c = p$. If CC is a set of *graphical consistency constraints*, we say that G satisfies CC, if G satisfies all constraints in CC.

A *graphical consistency constraint* (GCC) can be created choosing menu item New Atomic Constraint of pop-up menu GraGra. A GCC can contain more arbitrary conclusions. The default size of conclusions is equal to 1. A new conclusion can be created using menu item New Conclusion of pop-up menu GCC.
The left and right graphs and the morphism of a GCC are edited like a rule. The premise is only allowed to have constant values and variables as attribute values.
In AGG *graphical consistency constraints* are placed at the end of the grammar tree and marked by icon A.
We use the sample application *StateCharts* as an example of *graphical consistency constraints*.
In Figures 29 - 34 we can see a *graphical consistency constraint* with six conclusions. They describe the possible relations between two state nodes S or rather a state node S and state chart node SC.
Please note, in AGG a *graphical consistency constraint* with n conclusions is constructed as a set of n morphisms $P \rightarrow C$, where P is the same premise graph and C is always a different conclusion graph.

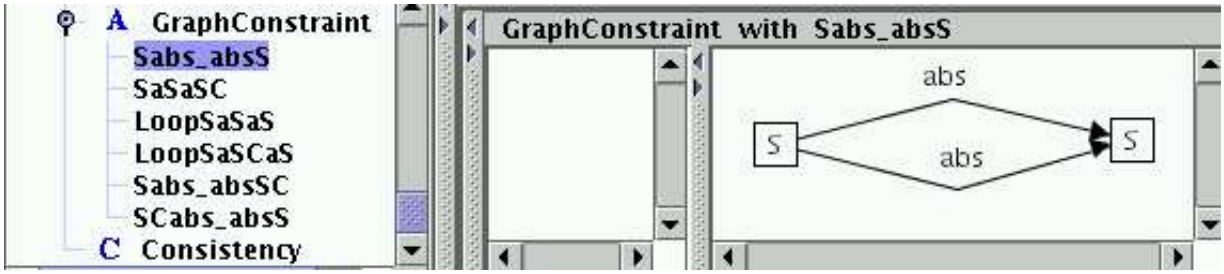


Figure 29: Sabs_absS conclusion of GraphConstraint

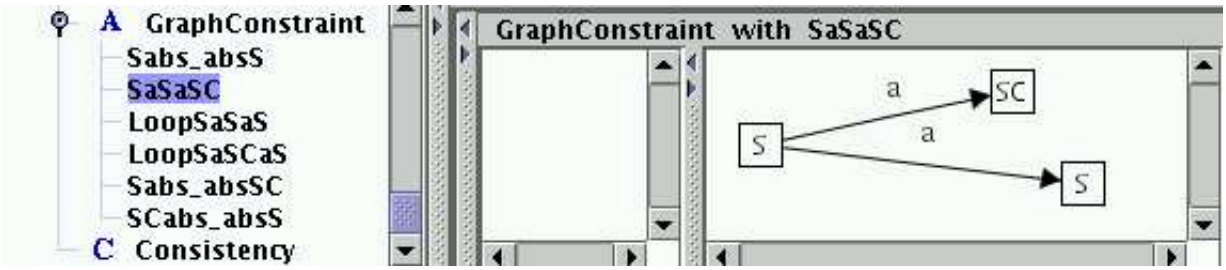


Figure 30: SaSaSC conclusion of GraphConstraint

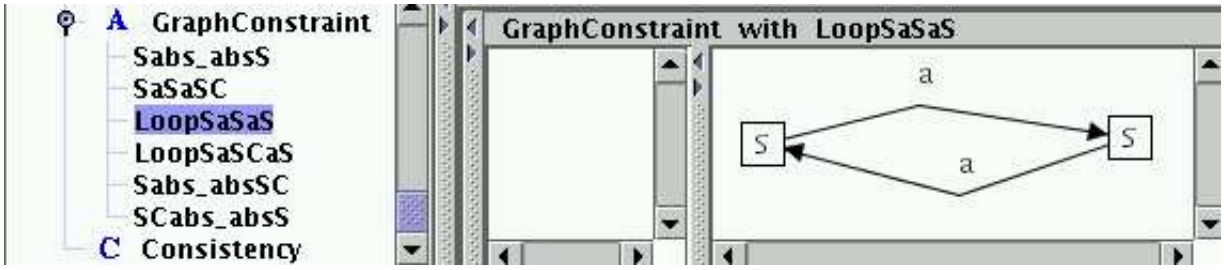


Figure 31: LoopSaSaS conclusion of GraphConstraint

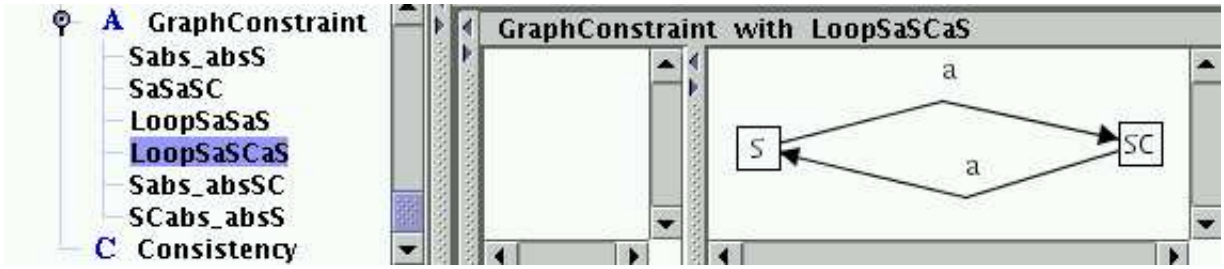


Figure 32: LoopSaSCaS conclusion of GraphConstraint

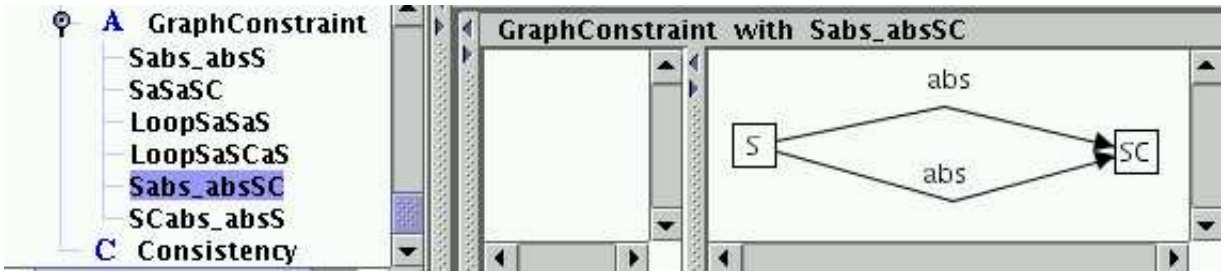


Figure 33: Sabs_absSC conclusion of GraphConstraint

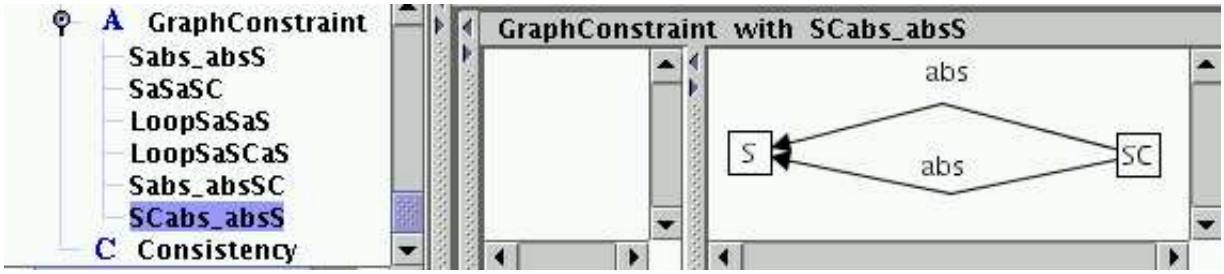


Figure 34: SCabs_absS conclusion of GraphConstraint

It is sufficient that at least one conclusion of a GCC is satisfied. We use a *consistency condition* (logical formula like this: $(\neg a)$) to make the GCC meaningful. We want to say, that this GraphConstraint is not allowed in graphs of *StateCharts*. *Consistency conditions* will be described in the next section.

3.1.2 Consistency Conditions

A *consistency condition* (CC) can be created by menu item New Constraint of pop-up menu GraGra. One can define a CC like a boolean formula using GCCs as variables.

A *consistency condition* can be defined in a formula editor which is available by menu item Edit of pop-up menu CC.

In AGG *consistency conditions* are placed at the end of the grammar tree and marked by icon C.

Figure 35: Editor of consistency condition

Please note, we use symbol ! for negation ¬.

To check the consistency of a graph grammar we can use :

- menu Analyzer / Consistency Check: *graphical consistency constraints* or *consistency conditions* of a selected grammar will be checked on its host graph,
- pop-up menu GraGra: *graphical consistency constraints* or *consistency conditions* of the invoked grammar will be checked on its host graph,
- pop-up menu of GCC: the invoked *graphical consistency constraint* will be checkt on the host graph,
- pop-up menu of CC: the invoked *consistency condition* will be checkt on the host graph.

3.1.3 Post Application Conditions

Using consistency conditions it is possible to generate *post application conditions* of a rule such that a graph grammar ensures consistency during rule application. [2]
Post application conditions can be generated:

- from all consistency conditions for all rules using menu item Create Post Conditions of pop-up menu GraGra;
- from all consistency conditions for a special espesial rule using menu item Create Post Conditions of pop-up menu Rule;
- from a special consistency condition for a special rule using menu item Create Post Conditions of pop-up menu Rule.

[2] *post application conditions* are not supported for graph grammars with inheritance relations

In AGG *post application conditions* are placed after the rule subtree and marked by icon P.

3.2 Critical Pair Analysis

AGG provides the analysis technique of *critical pair analysis*[3].

Critical pair analysis is known from term rewriting and used there to check if a term rewriting system is confluent. It has been generalized to graph rewriting. Critical pairs formalize the idea of a minimal example of a conflicting situation. From the set of all critical pairs we can extract the objects and links which cause conflicts or dependencies.

A *critical pair* is a pair of transformations $(p1, p2)$, with $p1(m1) : G => H1$ and $p2(m2) : G => H2$ which are in conflict, such that graph G being minimal. Roughly speaking, G is the gluing of the left-hand sides of the rules $p1$ and $p2$. It can be computed by overlapping $L1$ and $L2$ in all possible ways such that the intersection of $L1$ and $L2$ contains at least one item that is deleted or changed by one of the rules and both rules are applicable to G at their respective occurences.
The set of *critical pairs* represents precisely all *potential conflicts*. That means a *critical pair* $(p1, p2)$ exists iff, the application of $p1$ disables that of $p2$ or, vice versa.
After the computation of all *critical pairs*, the overall rule set can be structured into *conflict-free* and *conflicting* rules.

There are three reasons why rule applications can be *conflicting*. The first two are related to the graph structure while the last one concerns the graph attributes.

1. One rule application deletes a graph object which is in the match of another rule application.
2. One rule application generates graph objects in a way that a graph structure would occur which is prohibited by a negative application condition (NAC) of another rule application.
3. One rule application changes attributes being in the match of another rule application.

Two rule applications are in conflict if at least one of the conditions above is fulfilled. To find all conflicting rule applications, minimal critical graphs are computed to which rules can be applied in a conflicting way.
According to three conflict conditions mentioned above we distinguish three kinds of conflicts:

1. delete - use conflict: Here we consider overlapping graphs of the left-hand sides of two rules.

[3]Please note: *Critical Pair Analysis* is not implemented in case of *Typed Attributed Graph Transformation with Node Type Inheritance*

2. produce - forbid conflict: Here we consider overlapping graphs of the right-hand side of the first rule and a NAC-graph of the second rule. Such a NAC-graph consists of the left-hand side and graph objects prohibited by a negative application condition of a rule.

3. change - use attribute conflict: Here we consider overlapping graphs of the left-hand sides of two rules.
   In the case of a change - forbid attribute conflict we consider all overlapping graphs of the left-hand side of the first rule and a NAC-graph of the second rule.

Analogously, there are three reasons why rule applications can *depend* of one another. The first two are related to the graph structure while the last one concerns the graph attributes.

1. One rule application produces graph objects which are consumed by another rule application.

2. One rule application deletes graph objects in a way that a graph structure is deleted which is also prohibited by a negative application condition (NAC) of another rule application.

3. One rule application changes an attribute which is read by another rule application.

One rule application is dependent of another one if at least one of the conditions above is fulfilled. To find all dependent rule applications, minimal critical graphs are computed which apply rules in a depending way.
According to three dependencies conditions mentioned above we distinguish three kinds of dependencies:

1. produce - use dependency: Here we consider overlapping graphs of the right-hand side of the first rule and the left-hand side of the second rule.

2. delete - forbid dependency: Here we consider overlapping graphs of the left-hand side of the first rule and a NAC-graph of the second rule.

3. change - use attribute dependency: Here we consider overlapping graphs of the right-hand side of the first rule and the left-hand side of the second rule.
   In the case of a change - forbid attribute dependency we consider all overlapping graphs of the right-hand side of the first rule and a NAC-graph of the second rule.

Dependencies of two rules can be computed by inverting the first rule and finding its *conflicts* with the second rule and then by renaming

- delete - use conflict in produce - use dependency,

- produce - fobid conflict in delete - forbid dependency,

- change - use attribute conflict in change - use attribute dependency,

- change - forbid attribute conflict in change - forbid attribute dependency.

Additionally, in each case we consider the obvious matches and analyze the rule applications. All conflicting and depending rule applications we found are called *critical pairs*.

AGG provides the possibility to detect these two kinds of *critical pairs*:

- *conflicts* in the case of parallel dependent rule applications,
- *dependencies* in the case of sequential dependent rule applications.

The algorithm which computes *conflicts* and *dependencies* has been completely reworked. The new implementation is much faster and provides better results.

The graphical user interface (GUI) of *critical pair analysis* (*CPA*) was redesigned, too. In its new shape, the *CPA* GUI gives clear and sufficient information about critical pairs. The GUI is convenient to consider a critical pair in more detail. I.e. for each pair the two rules, all critical overlapping graphs and corresponding matches can be shown. The currently selected overlapping graph and the rule graphs used for this overlapping graph become highlighted in yellow. The overlapping graph objects (nodes and edges) are numbered to show embeddings of rule graphs. The critical graph objects are colored in green.

The menu Critical Pair Analysis in Fig. 36 is part of menu Analyzer. The availability of its items depends on the state of the critical pair analysis.

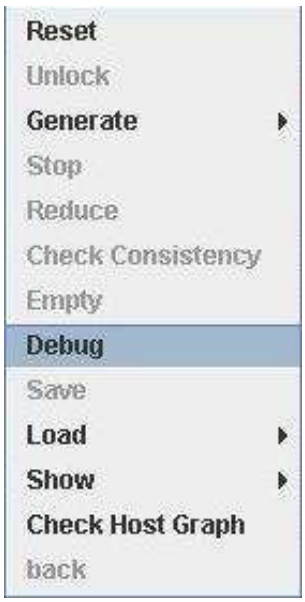


Figure 36: The Critical Pair Analysis menu

The items of menu Critical Pair Analysis are described below:

- Reset - Sets a selected grammar to be the grammar for *critical pair analysis*. This grammar will be locked during computation of critical pairs. This menu item should also be chosen after the current grammar has changed.

- Unlock - Allows to edit the grammar again.
  Please note: It is necessary to reset the grammar for *critical pair analysis* after a change of the grammar.

- Generate - Switches to the GUI of *critical pair analysis* and starts the generation of critical pairs.

  – Conflicts - Generates all parallel conflicts between rule applications
  – Dependencies - Generates all sequential dependencies of rule applications

  in minimal contexts.
  Critical pairs will be generated for all rules. After finishing generation you can select rule pairs to see results.
  Please note: It is not possible to edit the grammar after returning to the main AGG GUI. Use Unlock to make the grammar editable. Menu item Unlock becomes enabled after the generation has finished or has been stopped.

- Stop - Stops the generation process of critical pairs. The computation of the current rule pair is finished.

- Reduce[4]

- Check Consistency - All generated overlapping graphs of critical pairs will be checked against *graph consistency constraints*. The number of critical pairs is likely to decrease due to constraints.

- Empty - Removes all critical pairs.

- Debug - Allows to generate critical pairs step by step. After selecting a rule pair the computation of this pair is started.
  Please note: The next rule pair should be selected after the previous one has finished. It is not possible to edit the grammar after returning to the main AGG GUI. Use Unlock to make the grammar editable.

- Save - Saves computed critical pairs in a *.cpx* file. Saved critical pairs may be used for the parsing process. Using Save after Debug offers the possibility to save only a subset of critical pairs. That may be useful when the generation process is space and time intensive. The costs in space and time depend on the number of rules, on the size of their left hand sides and on the attributes.

- Load - Loads a critical pairs file. The name of such file ends with *.cpx*.

[4]Reduce is not used anymore with AGG 1.4.0

  - In This Window[5] - Show loaded critical pairs in main window.
  - In New Window[6] - Show loaded critical pairs in a new window.

- Show[7]
  - Conflicts - Shows conflicts table.
  - Dependencies - Shows dependencies table.
  - CPA Graph - Shows a relation graph based on computed conflicts/dependencies critical pairs.

- Check Host Graph - Allows to check which of the critical rules are applicable to a concrete input (host) graph. If two tables of critical pairs are available then the currently active (selected) table will be used to check the host graph.

- back - Returns to the main AGG GUI.

The options of the *critical pair analysis* are available in menu Preferences / Options... and shown in Figure 37.

The options are explaned here:

- Setting *Select algorithms for critical pairs* allows to choose the algorithm for *critical pair analysis*. *Conflict* critical pairs express *exclude* relations between rules. In the literature also *causal* dependencies of rule applications are considered. In this case, a *before* relation would be computed. The computation requested can be set by the upper combobox , choosing *conflicts* or *dependencies*.

- Setting *layered* can be used to optimize the generation process. If the graph grammar is *layered*, *critical pair analysis* works in the way that critical pairs are searched for rules in the same layer only.

- Setting *Layer to compute*[8] allows to set and compute critical pairs for a certain layer only or even for all layers.

- Setting *complete* means the complete computation of critical pairs. If this option is not set, the computation is stopped after the first critical pair has been found.

- Setting *consistent* forces the graph consistency check for all overlapping graphs.

- Setting *reduce*[9]

[5] Up to AGG 1.3.0 : In this Frame
[6] Up to AGG 1.3.0 : In separate Frame
[7] Show submenu is new with AGG 1.3.0
[8] Setting *Layer to compute* is new with AGG 1.4.0
[9] Setting reduce is not used anymore with AGG 1.4.0

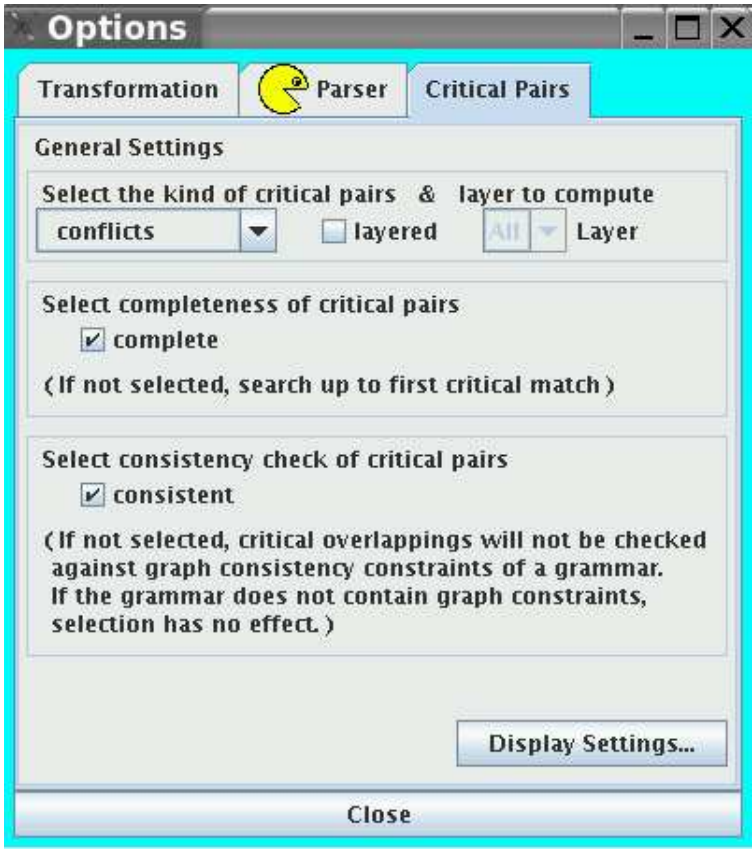


Figure 37: The options of the *critical pair analysis*

- *Display Settings...* can be used for a better overview of critical pairs. The maximal number of crticial pairs shown may be set, all pairs are shown by default. Furthermore, the initial window size for overlapping graphs may be set.

With AGG it is possible to show all conflicts and dependencies of rules within the so-called *CPA* graph. In this graph the nodes are rules, the red edges correspond to conflicts and the blue edges to dependencies between rules. The directed edges mean asymmetric and the undirected edges mean symmetric conflicts/dependencies between rules. A pop-up menu invoked on the background of the *CPA* graph allows to manipulate the graph view. Moreover, a pop-up menu on each entry of the critical pair table allows to hide/show appropriate node or edge(s) of the *CPA* graph.

The critical pair analysis GUI is shown in Figure 38 for example "StateCharts".
Clicking on a field of the certain rule pair in the table in the CPA panel has the following effect (the first rule is determined by the row, the second by the column):

1. The first rule and the second rule are shown in the CPA panel.
2. The computed critical graphs will be shown in the CPA panel.

3. The mappings between graph objects will be shown through equal numbers.
4. If no critical pairs exist, a message 'not critical' will be shown.

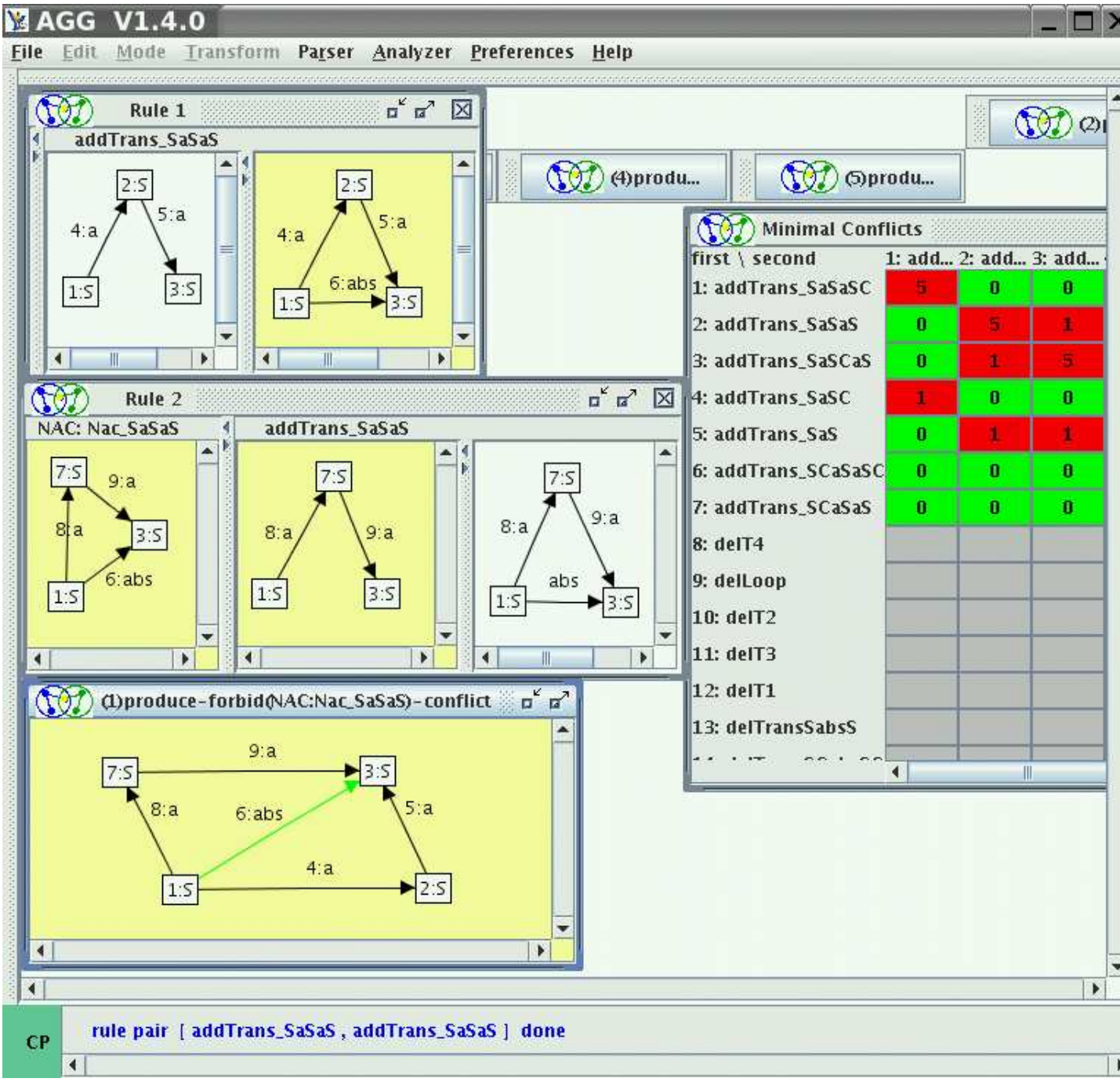


Figure 38: Critical Pair Analysis GUI

Figure 39 shows the critical pairs of rule addTrans_SaSaS with itself. This rule creats a new edge abs. Its NAC forbids multi-creation of this edge. So we can get a conflict situation during multi-application of this rule as shown by graphs in Figure 39.

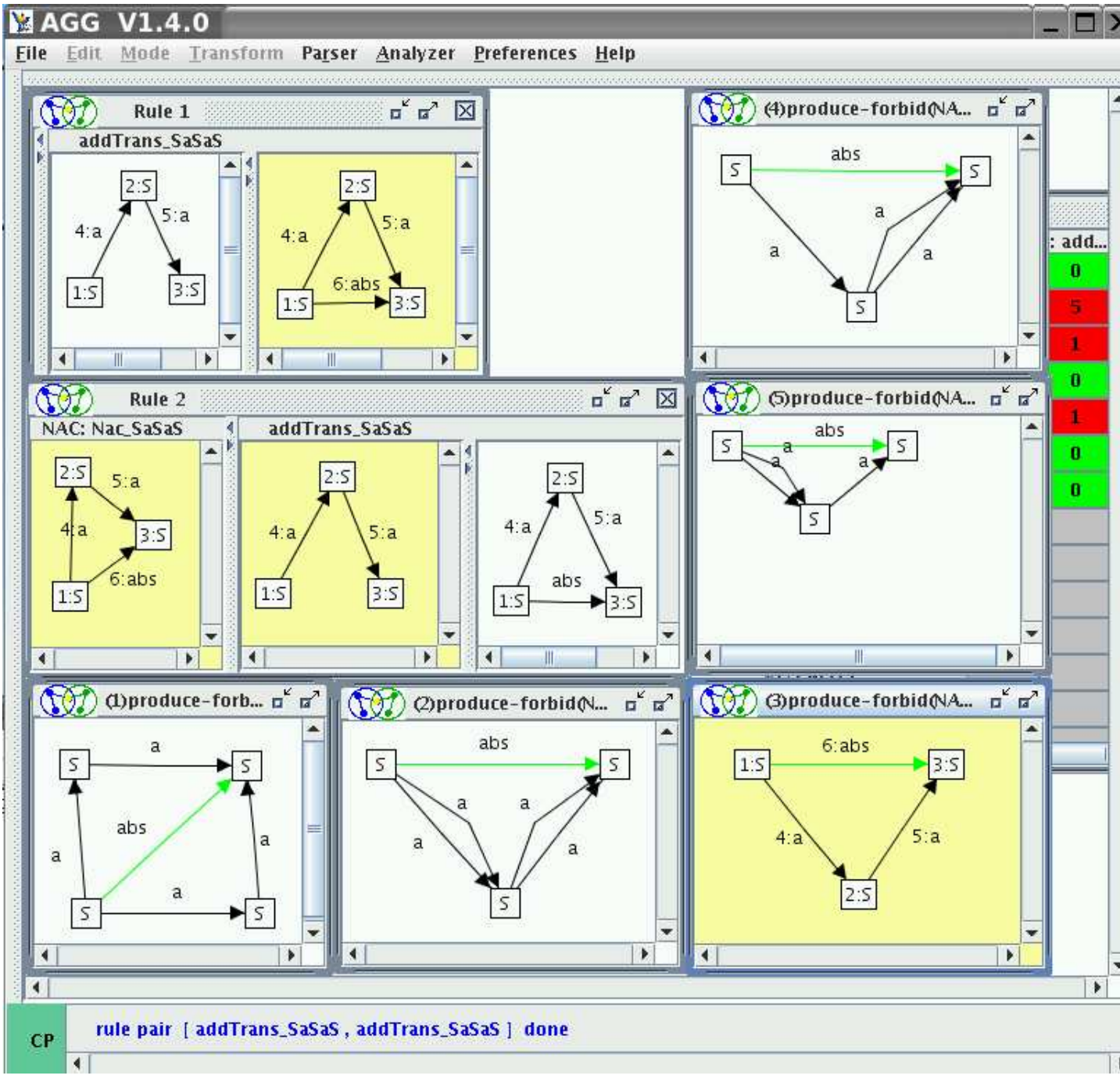


Figure 39: Critical Pairs of addTrans_SaSaS with itself

Another critical rule pair of this grammar is shown in Figure 40.

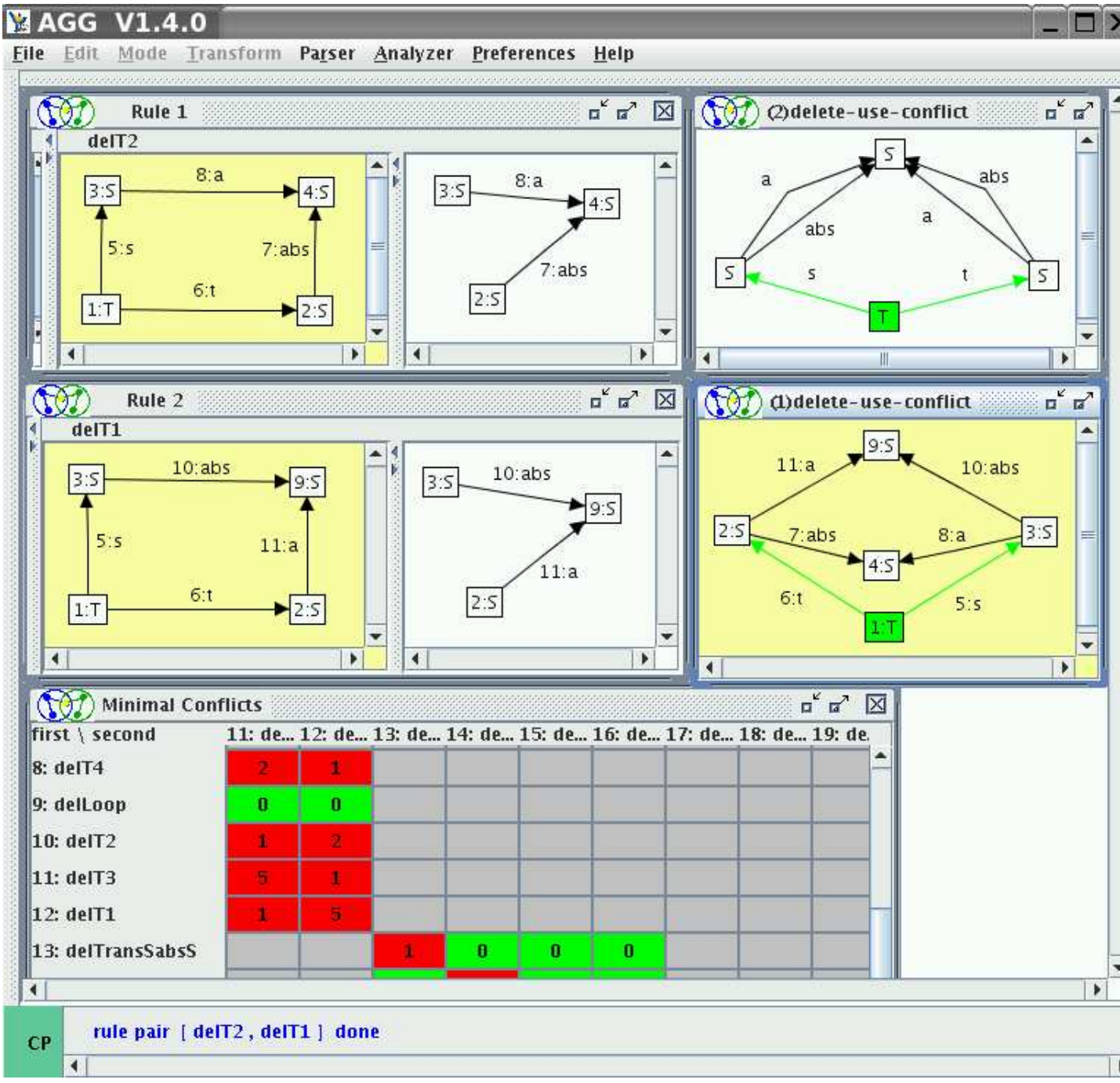


Figure 40: Critical Pairs of delT2 with delT1

Here we can see critical graph objects. The conflict occurs, because the first rule deletes

objects which the second rule needs.

In Figure 41 the complete set of generated critical pairs is shown as a table with red (critical) and green (non-critical) fields of rule pairs.

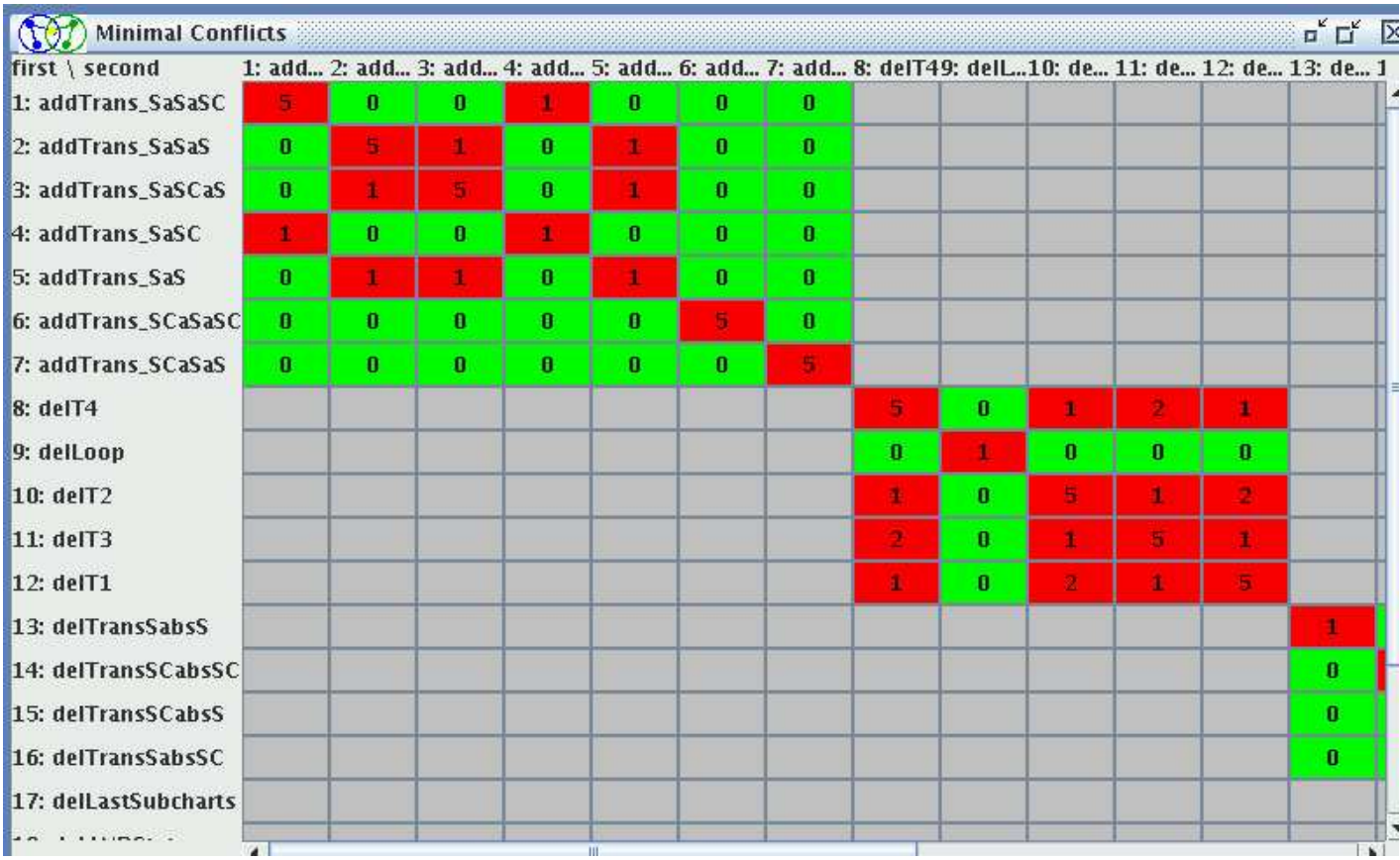


Figure 41: Critical Pairs of StateCharts

Critical pair analysis for grammar rules can be more efficient, if a type graph is defined containing multiplicity constraints. Upper bounds of multiplicity constraints are used to reduce the set of critical pairs by throwing out the meaningless ones. Dependent on the graph grammar, the number of critical pairs can be reduced drastically by the use of multiplicity constraints.
Moreover, the set of critical pairs can be reduced considerably by defining graph constraints. In the reduced set there are critical pairs with consistent overlapping graphs only.
Please note: Not all kinds of graph constraints are used for reduction, but only those which forbid the existence of graph structures.

Figure 42 shows the type graph of the *StateCharts* grammar.

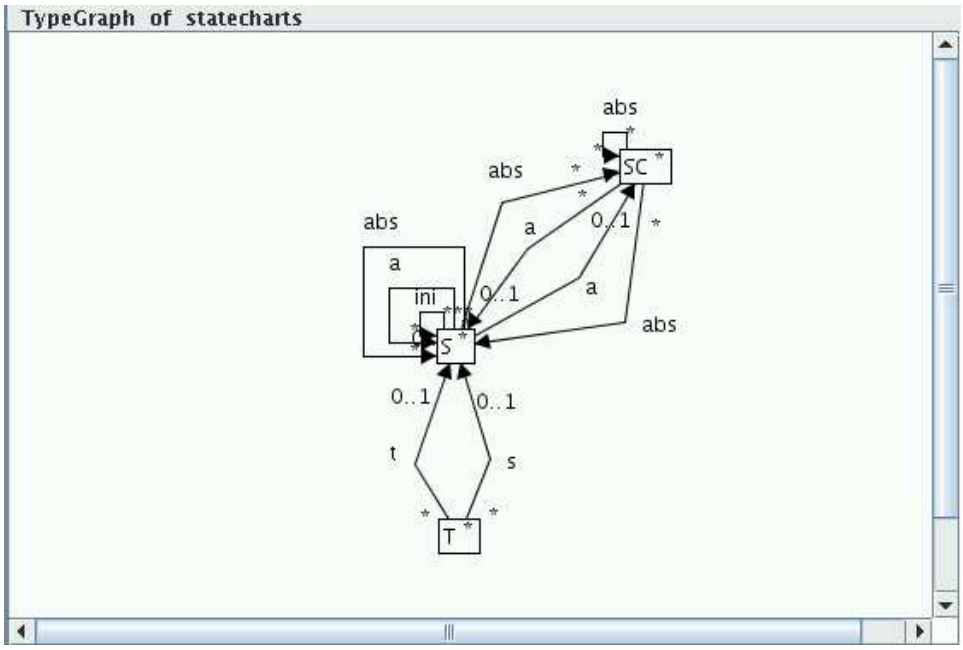


Figure 42: Type graph of the *StateCharts* grammar

Figure 43 shows critical pairs of the *StateCharts* grammar with the type graph in Figure 42 and graph constraints in Figures 29 - 34.
Please note: the *critical pair analysis* may take time, even for one particular rule pair.

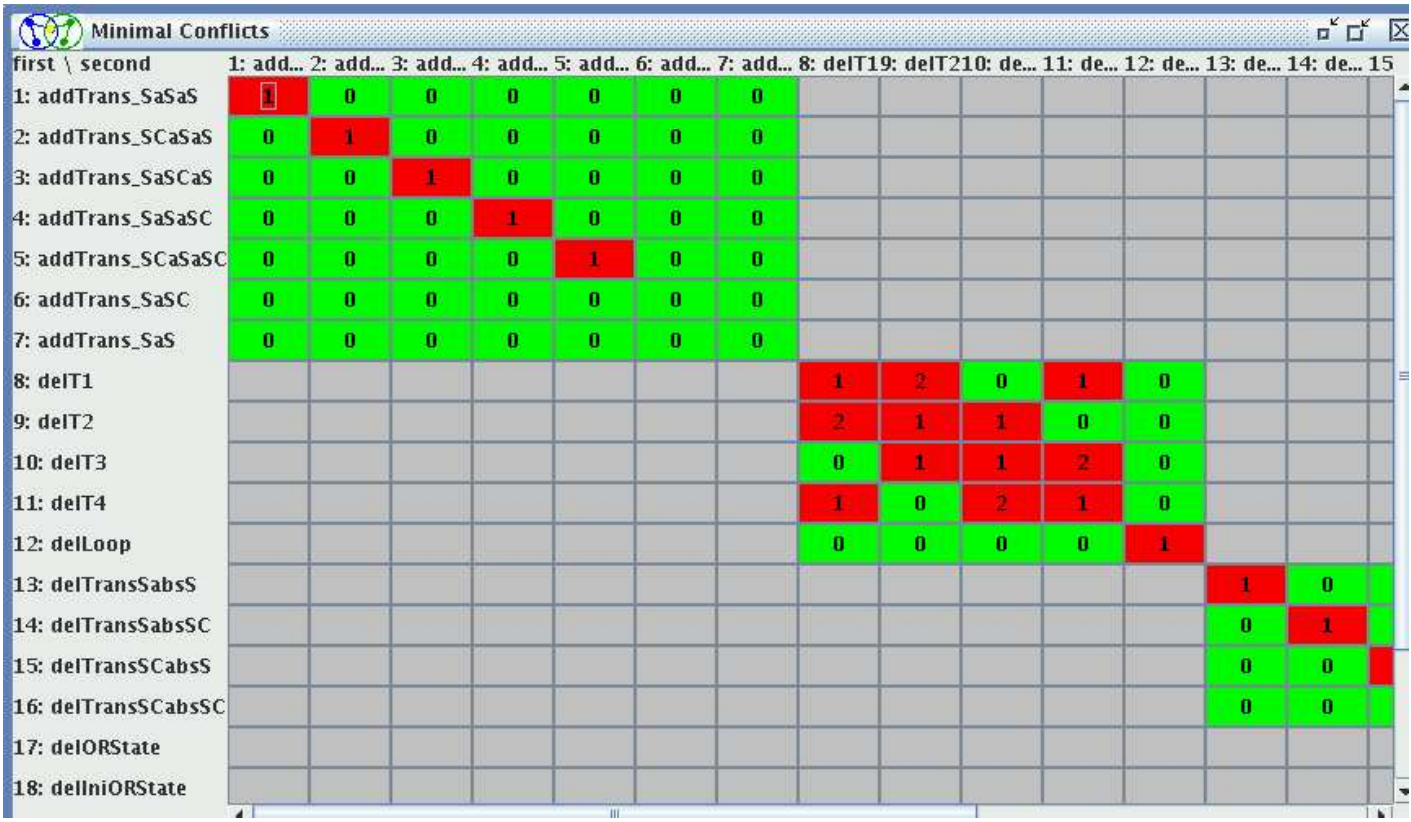
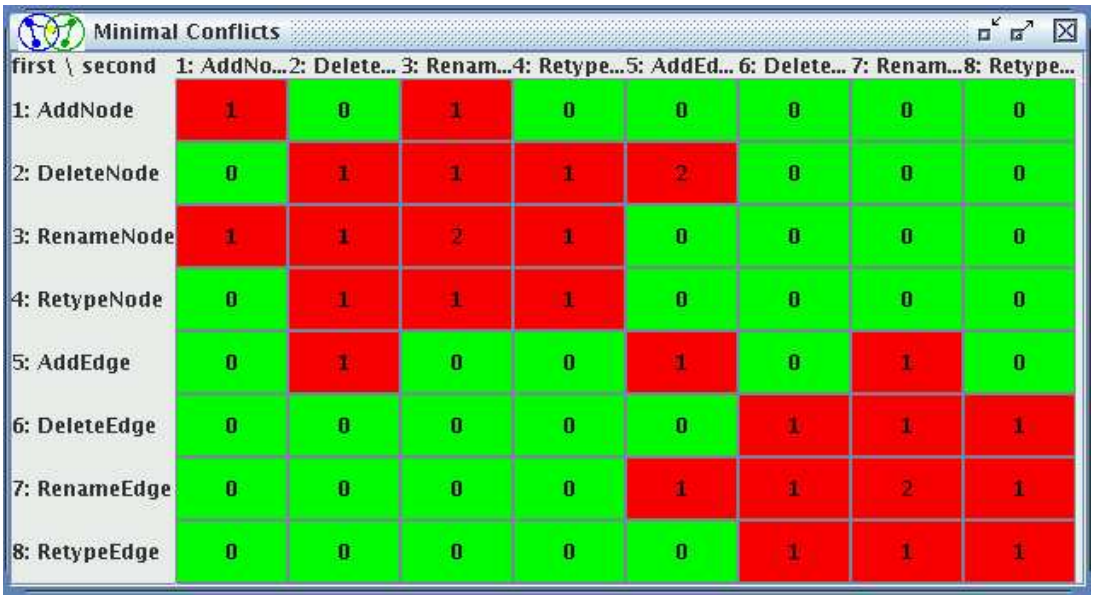


Figure 43: Critical Pairs of StateCharts with a type graph and graph constraints

As already said above, critical pairs show potential parallel conflicts and sequential dependencies between transformations. The computation of conflicts is started by menu item Generate/Conflicts, the computation of dependencies by menu item Generate/Dependencies.

Now we want to use a refactorings example *"BasicGraph"* to show not only these two kinds of conflicts, but also to introduce a graph representation based on computed critical pairs. The grammar *"BasicGraph.ggx"* can be found in the examples folder *Refactorings* when downloading AGG . This example was introduced in the PhD thesis of Tom Mens: "*A Formal Foundation for Object-Oriented Software Evolution.*" (PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 1999.) *"BasicGraph"* is a very general case study. The models that can be represented by this grammar are directed, attributed, typed graphs. The rules are very simple, but allow to make arbitrarily complex changes to any given graph structure. There are the following rules : *AddNode*, *AddEdge*, *DeleteNode*, *DeleteEdge*, *RenameNode*, *RenameEdge*, *RetypeNode* and *RetypeEdge*. Figure 44 shows conflicts between the rule applications.



first \ second	1: AddNo...	2: Delete...	3: Renam...	4: Retype...	5: AddEd...	6: Delete...	7: Renam...	8: Retype...
1: AddNode	1	0	1	0	0	0	0	0
2: DeleteNode	0	1	1	1	2	0	0	0
3: RenameNode	1	1	2	1	0	0	0	0
4: RetypeNode	0	1	1	1	0	0	0	0
5: AddEdge	0	1	0	0	1	0	1	0
6: DeleteEdge	0	0	0	0	0	1	1	1
7: RenameEdge	0	0	0	0	1	1	2	1
8: RetypeEdge	0	0	0	0	0	1	1	1

Figure 44: Minimal conflicts between rules of *"BasicGraph"*

We want to get a closer look at the rule pair *DeleteNode* and *AddEdge*. The rules and two *delete-use* conflict situations are shown in Fig. 45. It is clear that the node to delete could be a source or a target node of a new edge to add. After the rule *DeleteNode* is applied, the rule *AddEdge* cannot be applied to previous two nodes.

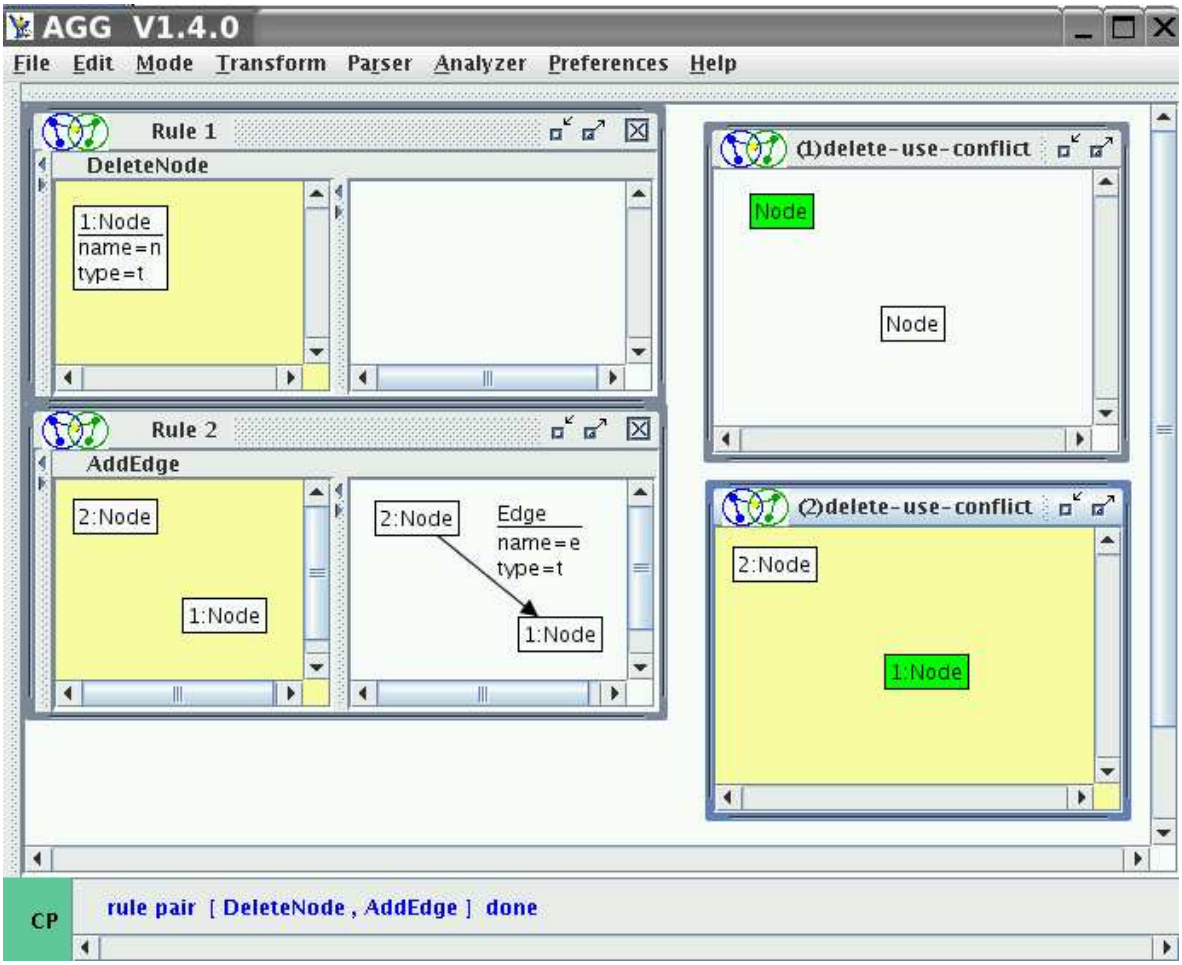


Figure 45: Conflict of *DeleteNode* with *AddEdge*

Another example in Fig. 46 shows two attribute conflicts of rule *RenameNode* with itself. In the case of a change - use attribute conflict this rule changes the name of a node which is in the match of its other application. In the case of a change - forbid attribute conflict the new name of the node can occur in a graph structure that is prohibited by a negative application condition of the rule and its application is not possible anymore.

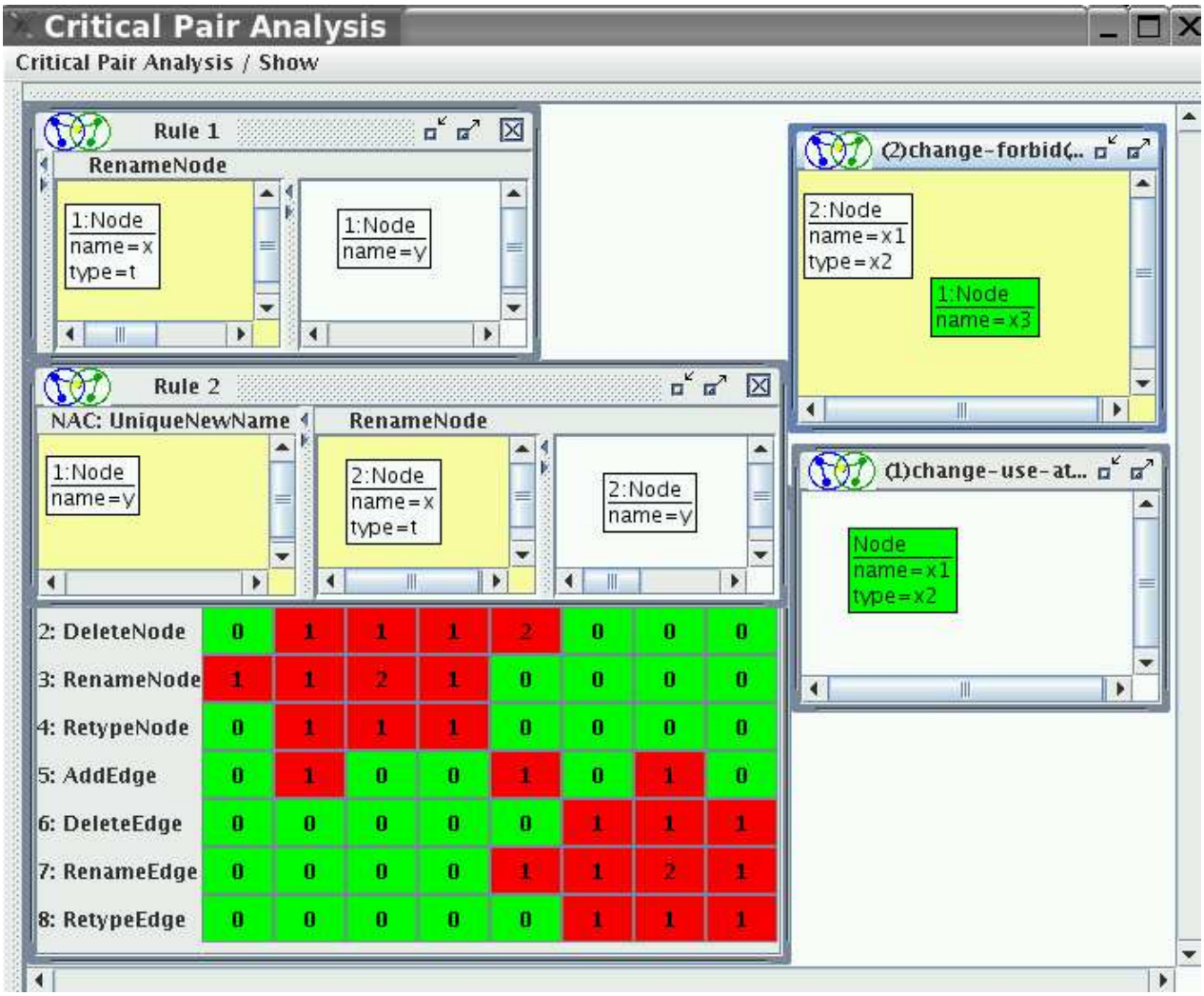
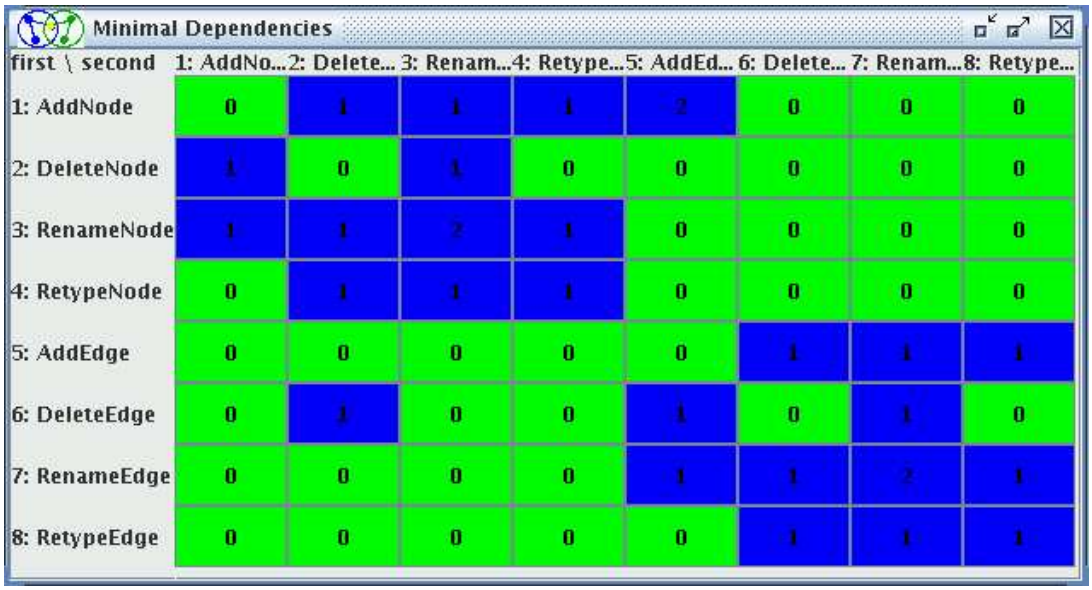


Figure 46: Attribute conflicts of *RenameNode* with itself

The table of critical pair analysis in Fig. 47 shows all those critical pairs reporting sequential dependencies between rule applications.



first \ second	1: AddNo...	2: Delete...	3: Renam...	4: Retype...	5: AddEd...	6: Delete...	7: Renam...	8: Retype...
1: AddNode	0	1	1	1	2	0	0	0
2: DeleteNode	1	0	1	0	0	0	0	0
3: RenameNode	1	1	2	1	0	0	0	0
4: RetypeNode	0	1	1	1	0	0	0	0
5: AddEdge	0	0	0	0	0	1	1	1
6: DeleteEdge	0	1	0	0	1	0	1	0
7: RenameEdge	0	0	0	0	1	1	2	1
8: RetypeEdge	0	0	0	0	0	1	1	1

Figure 47: Minimal dependencies between rules of *"BasicGraph"*

Now we have a look at the rule pair *AddNode* and *AddEdge*. The rules and two *produce-use* dependency situations are shown in Fig. 48. As we can see, first a special node should be added and then the desired edge.

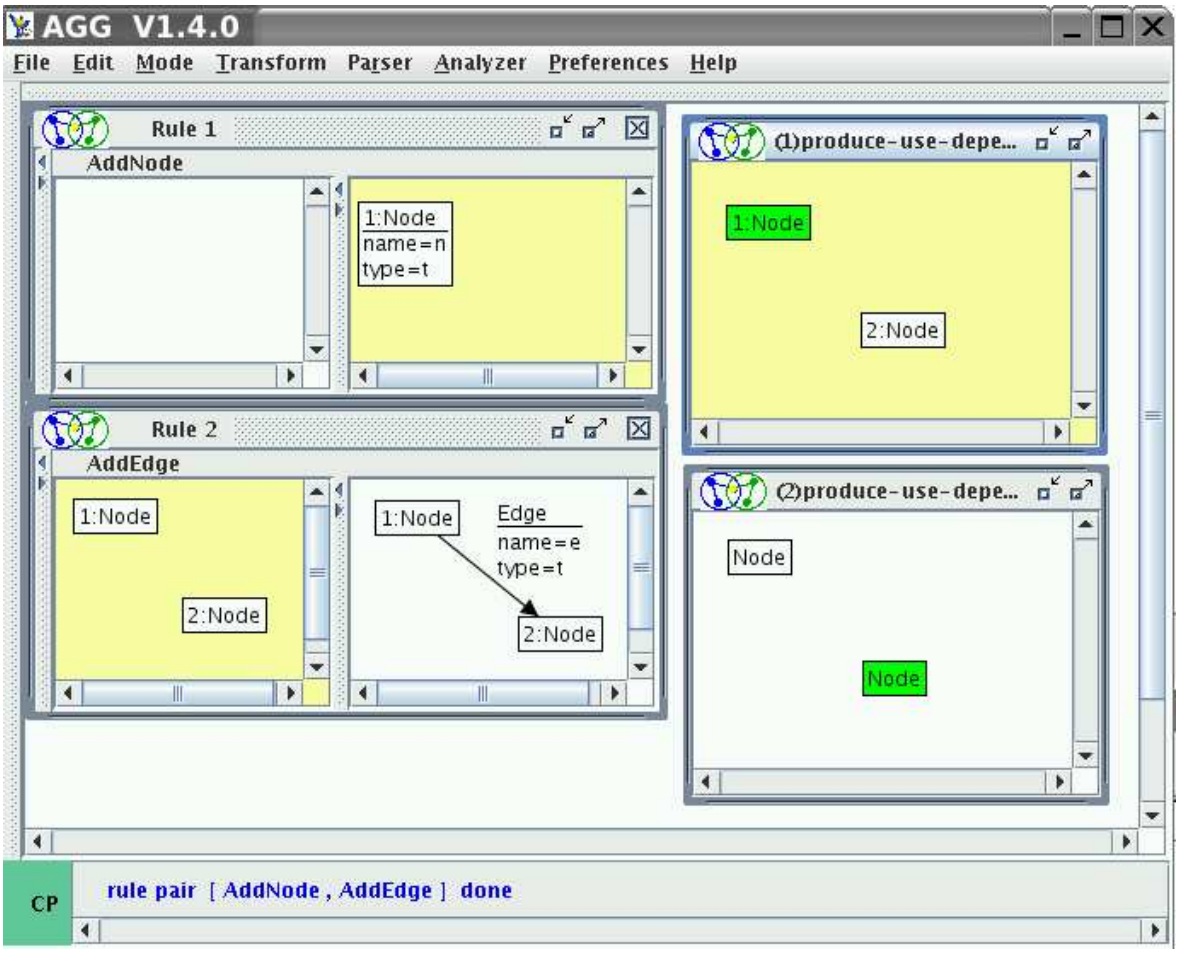


Figure 48: Sequential dependency *AddEdge* on *AddNode*

An example of a change - use attribute dependency is shown in Fig. 49. The application of rule *RetypeNode* may read the value of the attribute *name* after the application of rule *RenameNode* has changed this attribute in suitable way.

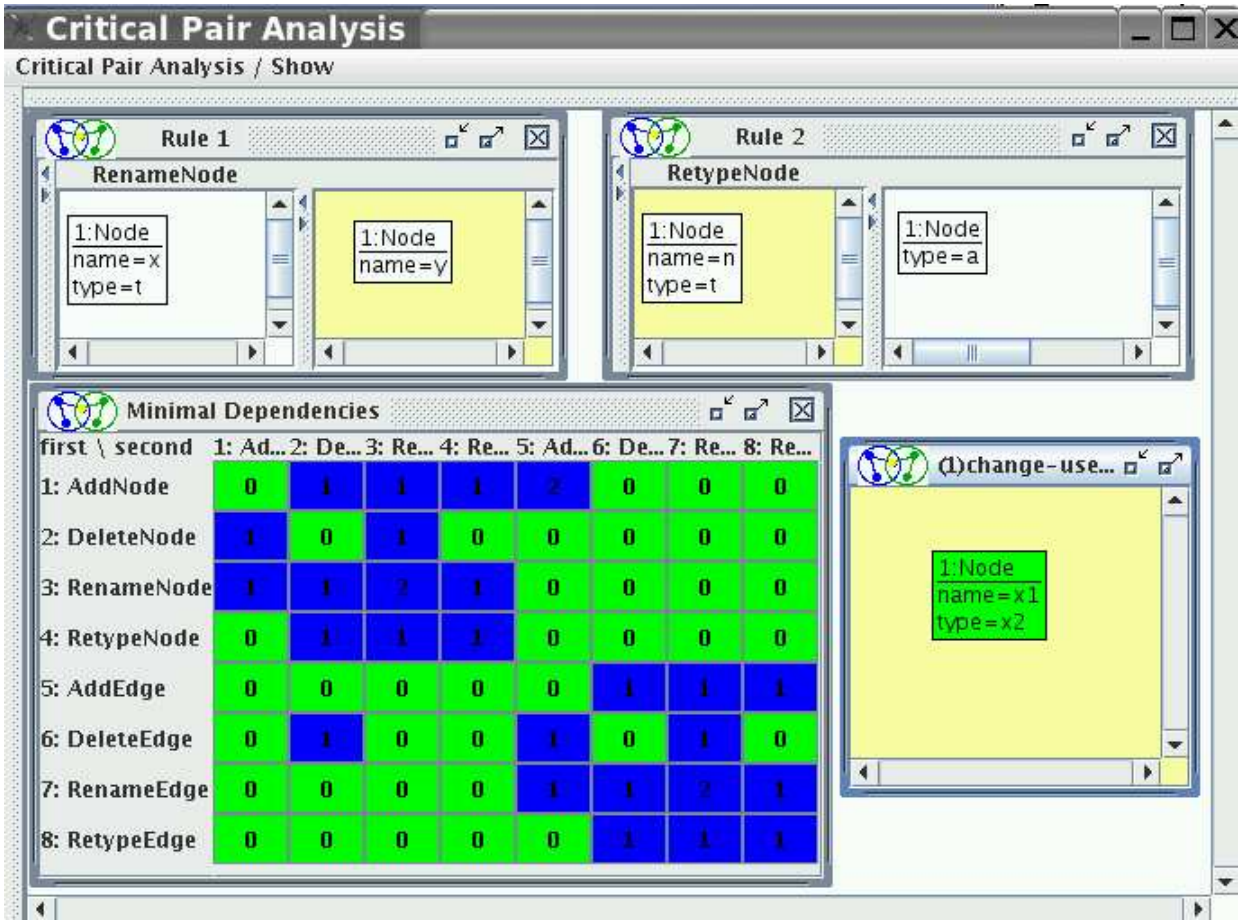


Figure 49: Attribute dependency of *RetypeNode* on *RenameNode*

As said above, AGG provides the possibility to visualize conflicts and dependencies within the so-called CPA graph. This graph is automatically computed by considering all critical pairs for conflicts and dependencies. The CPA graph for example *"BasicGraph"* is shown in Fig. 50.

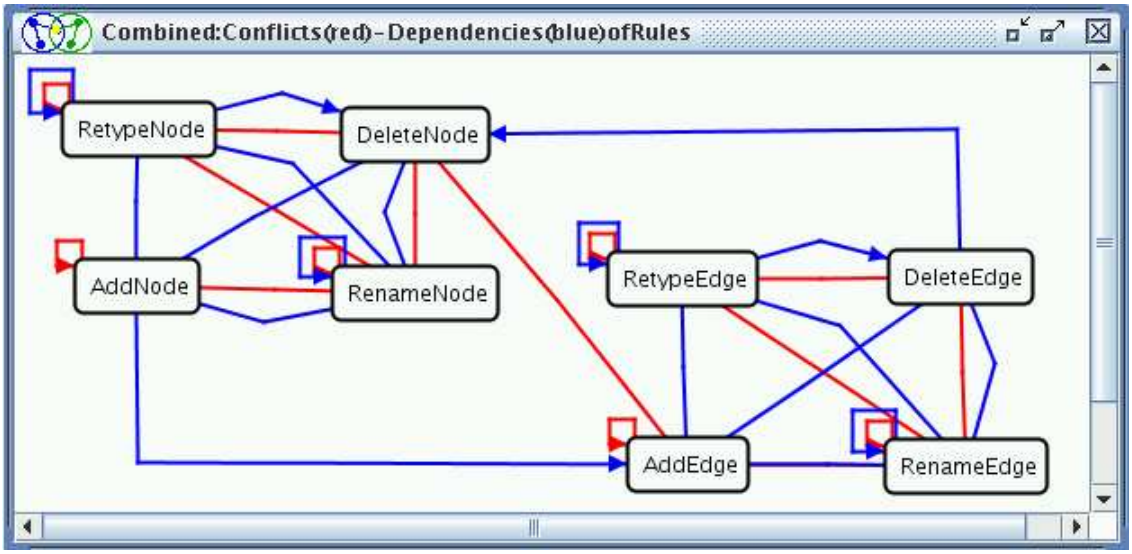


Figure 50: CPA graph for *"BasicGraph"*

Using the pop-up menu of this graph we can get a view on conflicts or dependencies only. Such views on the CPA graph are shown in Figures 51 - 52.

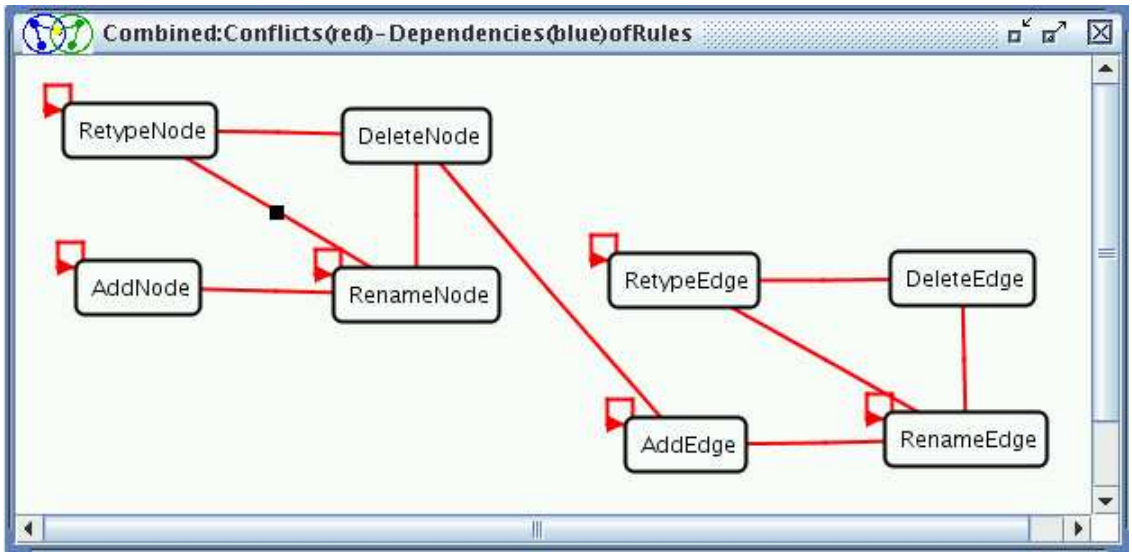


Figure 51: CPA graph for *"BasicGraph"* with conflicts only

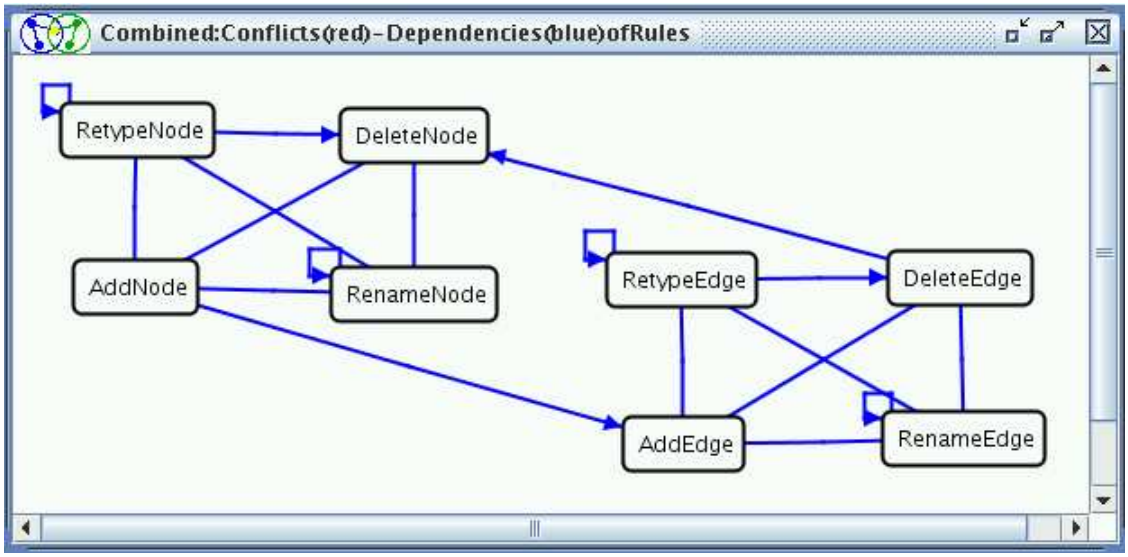


Figure 52: CPA graph for *"BasicGraph"* with dependencies only

The CPA graph can be very complex if the given rules have many conflicts and dependencies. In order to get a better understanding of all conflicts and dependencies it is useful to hide certain edges and nodes dynamically. This can be done by right-clicking on a node or an edge and choosing item *Hide Node/Edge* of the appearing pop-up menu. Another possibility to manipulate the CPA graph is by using pop-up menus on the entries of the conflict and dependency tables and choosing item *Hide Relation* or *Hide Rule*. A certain rule has to be hidden in each table, before it becomes hidden in the CPA graph.

Analysing the CPA graph, the user can get some help about the order in which the transformations should be performed. For example, Fig. 52 shows that *DeleteEdge* should be performed before *DeleteNode*. On the other hand, in Fig. 51 one can see that the rules *DeleteNode*, *RenameNode*, *RetypeNode* and *AddEdge* are all in symmetrical conflict with each other.

The automatic conflict resolution analysis in AGG is left to future work.
Critical pair analysis is described in:

- "Confluence of Typed Attributed Graph Transformation Systems" by Reiko Heckel, Jochen Malte Küster and Gabriele Taentzer `http://tfs.cs.tu-berlin.de/agg/critical\_pairs/ConfluenceOfTAGTS\_paper.pdf`
- "Entwicklung und Implementierung eines Parsers für visuelle Sprachen basierend auf kritischer Paaranalyse" Diploma thesis of Thorsten Schultzke (in German) `http://tfs.cs.tu-berlin.de/agg/Diplomarbeiten/ThorstenDiplomarbeit.ps`

Critical pair analysis is used in:

- "Detection of Conflicting Functional Requirements in a Use Case-Driven Approach" by Jan Hendrik Hausmann, Reiko Heckel and Gabriele Taentzer (`http://tfs.cs.tu-berlin.de/agg/critical\_pairs/paperfinal.pdf`).
- "Efficient Parsing of Visual Languages based on Critical Pair Analysis and contextual Layered Graph Transformation" by P. Bottoni, A. Schürr and G. Taentzer (`http://tfs.cs.tu-berlin.de/agg/parser/BST-long.ps`).
- "Detecting Structural Refactoring Conflicts Using Critical Pair Analysis" by Tom Mens, Gabriele Taentzer and Olga Runge (`http://www.cs.tu-berlin.de/~gabi/gMTR04.pdf`).

3.3 Graph Parser

AGG provides a graph parser which is able to check, if a given graph belongs to a certain graph language determined by a graph grammar. In formal language theory, this problem is known as the *membership problem*. Here, the membership problem is lifted to graphs. This problem is undecidable for graph grammars in general, but for restricted classes of graph grammars it is more or less efficiently solvable.
Usually, a graph grammar is given to generate all graphs of a graph language. For parsing, all rules of the graph grammar have to be inverted. Applying the inverted rules in a right way, all graphs of the corresponding graph language can be reduced to the start graph of the grammar.
In AGG not all kinds of rules can be automatically inverted. That's why the graph parser expects already a so-called parse grammar containing reducing parsing rules and a stop graph. Given an arbitrary host graph, AGG tries to apply the parsing rules in a way that the host graph is reduced to the stop graph. If this is possible, the host graph belongs to the determined graph language and there is a derivation sequence from the host graph to the stop graph, otherwise the host graph does not belong to the graph language.

AGG offers three different variants of a parsing algorithm being based on backtracking. The parser builds up a derivation tree of possible reductions of the host graph. The leaves of this tree are either dead ends, i.e. leave graphs where no rule can be applied anymore but non-isomorphic to the stop graph or the stop graph. In this case, the parsing was successful. Since simple backt racking has exponential time complexity, the simple backtracking parser is accompanied by two further parser variants exploiting *critical pair analysis* for rules.

Critical pair analysis can be used to make parsing of graphs more efficient: decisions between conflicting rule applications are delayed as far as possible. This means that *non-conflicting* rules are applied first to reduce the graph as much as possible. Afterwards, the *conflicting* rules are applied, first in non-critical situations and when this is not possible, in critical ones. In general, this optimization reduces the derivation tree constructed, but

does not change the worst case complexity.

The parsing process might not terminate, therefore *layering conditions* have to be defined to guarantee termination of the parsing process. Parsing of *non-layered* or *layered* graph grammars can optionally be based on *critical pair analysis*.

The graphical user interface of the graph parser allows to consider the parsing process step-by-step. The parser menu and options are described below.

The parser menu is available from the menu bar and contains the following items:

- Open - Opens a user dialog to initialize the parser and then changes to the parser GUI.
- Start - Starts the parsing process.
- Stop - Stops the parsing process.
- back - Returns to the main AGG GUI.

The options for the parser are available in menu Preferences / Options.... They are shown in Figure 53 and described below.

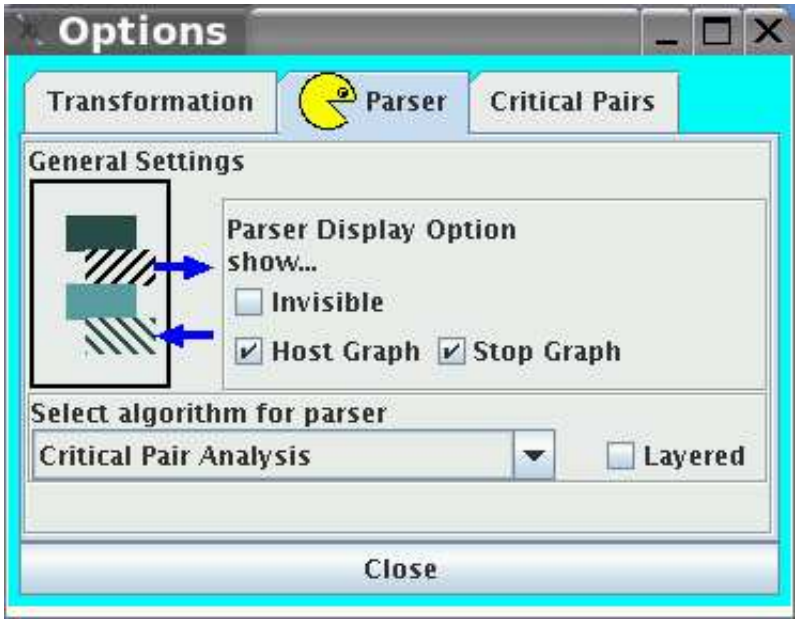


Figure 53: The parser options

- *Parser Display Option* is used to configure the visualization of a parsing process,
- *Invisible* can be used to make the *host graph* and/or the *stop graph* invisible,

- *Host graph* - the host graph is visible,
- *Stop graph* - the stop graph is visible,
- *layered* means layered parsing process: First all rules of one layer are applied as long as possible before the rules of the next layer are applied,
- *Select algorithm for parser* concerns the variant of parsing algorithm to be used. Dependent on the use of critical pairs the user may choose:
  - *Critical Pair Analysis*[10] uses critical pair analysis to check if potentially conflicting rules really exclude rule applications in the actual host graph and applies non-conflicting rules first.
  - *Semi-optimized backtracking*[11] applies non-conflicting rules first.
  - *Backtracking without optimization* does not use critical pairs.

Before the parsing process can start, we have to set the needed data in the Starting Parser dialog shown in Figure 54. This dialog is invoked by item Open of menu Parser.

The items and buttons of the starting dialog are exlpained bellow:

- Select Host Graph waits for selecting or loading a grammar which contains a host graph.
- Select Stop Graph waits for selecting or loading a parsing grammar which contains the parsing rules and a stop graph.
- Select Critical Pairs has a special meaning. The critical pairs are connected with their parsing rules. Therefore, the parsing grammar has to be selected to generate critical pairs. If the critical pairs are already generated and stored, they can be loaded.
- The Next button moves the selector to the next item. If the initialization is completed, the Finish button appears.
- The Finish button closes the user dialog and opens the parser GUI.
- The Cancel button cancels the user dialog.

Figure 55 shows the Parser GUI for example "Statecharts".
Using menu item Parser / Start start the parsing process.

Graph parsing in the context of visual language parsing is described in:

[10] This algorithm is not implemented in case of *Typed Attributed Graph Transformation with Node Type Inheritance*

[11] This algorithm is not implemented in case of *Typed Attributed Graph Transformation with Node Type Inheritance*

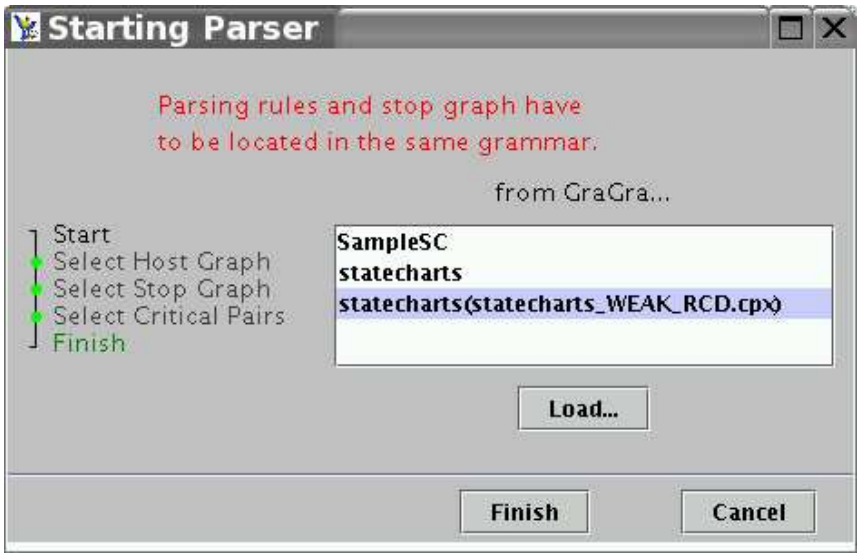


Figure 54: The starting parser dialog

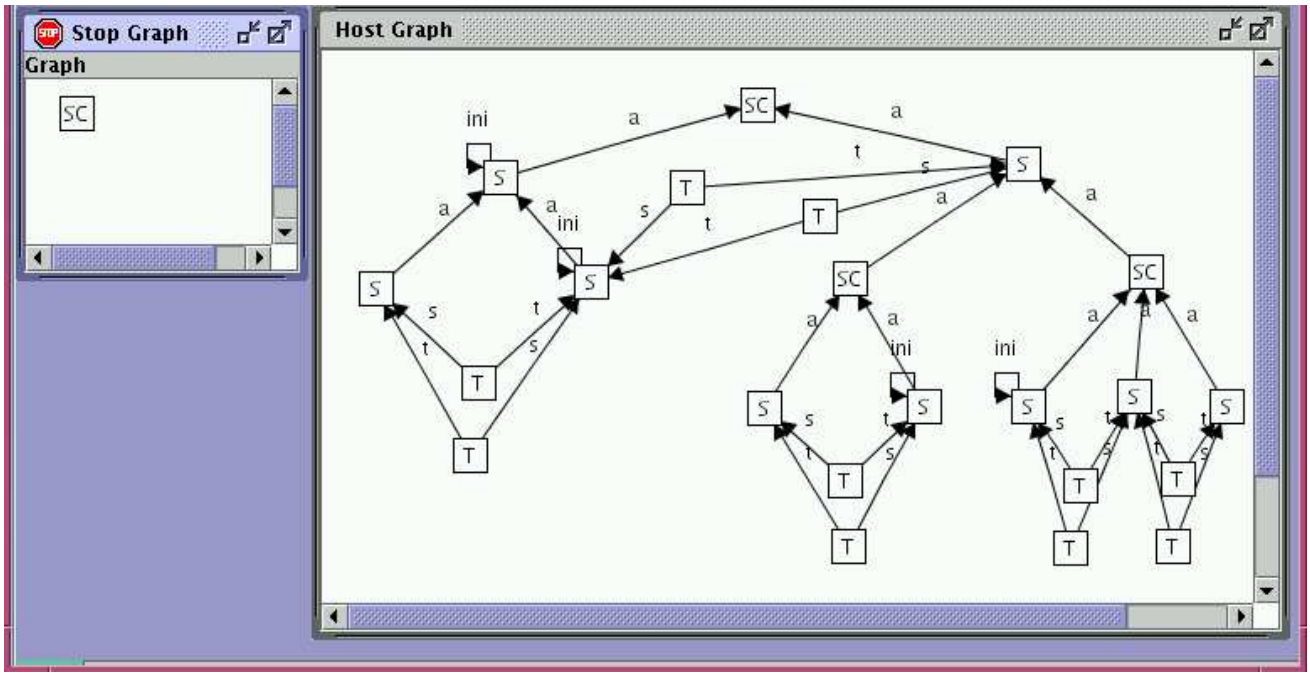


Figure 55: The parser GUI

- "Application of Graph Transformation to Visual Languages" by R. Bardohl, M. Minas, A. Schürr and G. Taentzer (`http://www.cs.tu-berlin.de/gabi/gBMST98.ps.gz`).

- "Efficient Parsing of Visual Languages based on Critical Pair Analysis and contextual Layered Graph Transformation" by P. Bottoni, A. Schürr and G. Taentzer (`http://www.cs.tu-berlin.de/agg/parser/BST-long.ps`).

More about the facilities of the parser and the critical pair analysis in AGGis described in the diploma thesis of Thorsten Schultzke (in German): "Entwicklung und Implementierung eines Parsers für visuelle Sprachen basierend auf kritischer Paaranalyse" (`http://tfs.cs.tu-berlin.de/agg/Diplomarbeiten/ThorstenDiplomarbeit.ps`).

3.4 Termination Criteria for LGTS

In AGG termination criteria are implemented for *Layered Graph Transformation Systems* (*LGTS*) (see section 1.10). In general, termination is undecidable for graph grammars. But if the graph grammars meet suitable termination criteria, we can conclude that they are terminating. The criteria we propose are based on assigning a *layer* to each rule, node and edge type.
For termination, we define layered graph grammars with deletion and non-deletion layers. Termination criteria are expressed by deletion and nondeletion layer conditions.

A graph grammar is called *layered graph grammar (LGG)* if for each rule r we have a rule layer $rl(r) = k$ and for each label l a creation layer $cl(l)$ and a deletion layer $dl(l)$. The following conditions have to be satisfied for all rules:

(1) *Deletion Layer Conditions*
If k is a deletion layer, then

1. each rule r decreases the number of graph items, or
2. each rule r decreases the number of graph items of one special type.

(2) *Deletion Layer Conditions*
If k is a deletion layer, then

1. r deletes at least one item,
2. $0 <= cl(l) <= dl(l) <= n$ for all l,
3. if r deletes l then $dl(l) <= rl(r)$,
4. if r creates l then $cl(l) > rl(r)$.

(3) *Nondeletion Layer Conditions*
If k is a nondeletion layer, then

1. r is nondeleting, i.e. $r : L- > R$ is total and injective,
2. r has NAC $n : L- > N$ with $n' : N- > R$ injective s.t. n' o $n = r$,
3. if l occurs in L then $cl(l) <= rl(r)$,
4. if r creates l then $cl(l) > rl(r)$.

The termination criterion (1) and (2) are a general termination criteria of typed layered graph transformation systems.
The termination criterion (3) is defined for typed graph transformation with injective rules, injective matches and injective negative application conditions (NACs).

The *deletion layer conditions* (2) ensure that the last creation of an element of a certain type always precedes the first deletion of an element of the same type.
On the other hand, *nondeletion layer conditions* (3) ensure that if an element of a certain type occurs in the LHS of a rule then all elements of the same type were already created in this or a previous layer. New elements have to have creation layers larger than the layer of the rule which creates them.

A layered graph grammar with deletion and nondeletion layers terminates, if for each layer the deletion or nondeletion layer conditions defined above are satisfied.
The rule layers can be set or generated. The creation and deletion type layers will be generated automatically s.th. for each layer one set of layer conditions is satisfied, if this is possible.

The graphical user interface to define rule, creation and deletion layers is available by menu item Analyzer/Termination Check and shown in Figure 56. There, the rules and user-defined rule layers of our example *StateCharts* are shown.

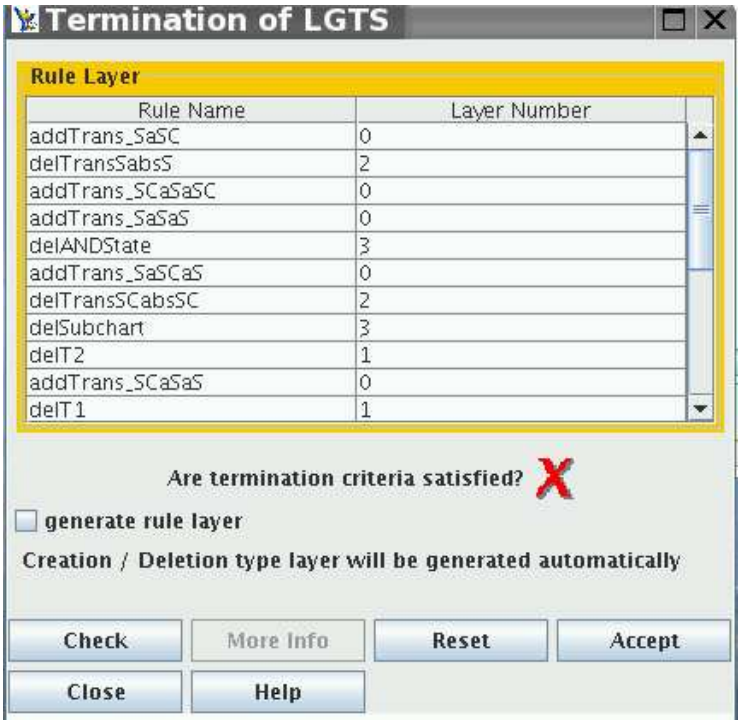


Figure 56: Termination dialog

Figure 57 shows the generated creation and deletion layers for types in addition. Moreover, we can see that the termination criteria are fulfilled by these layer assignments.

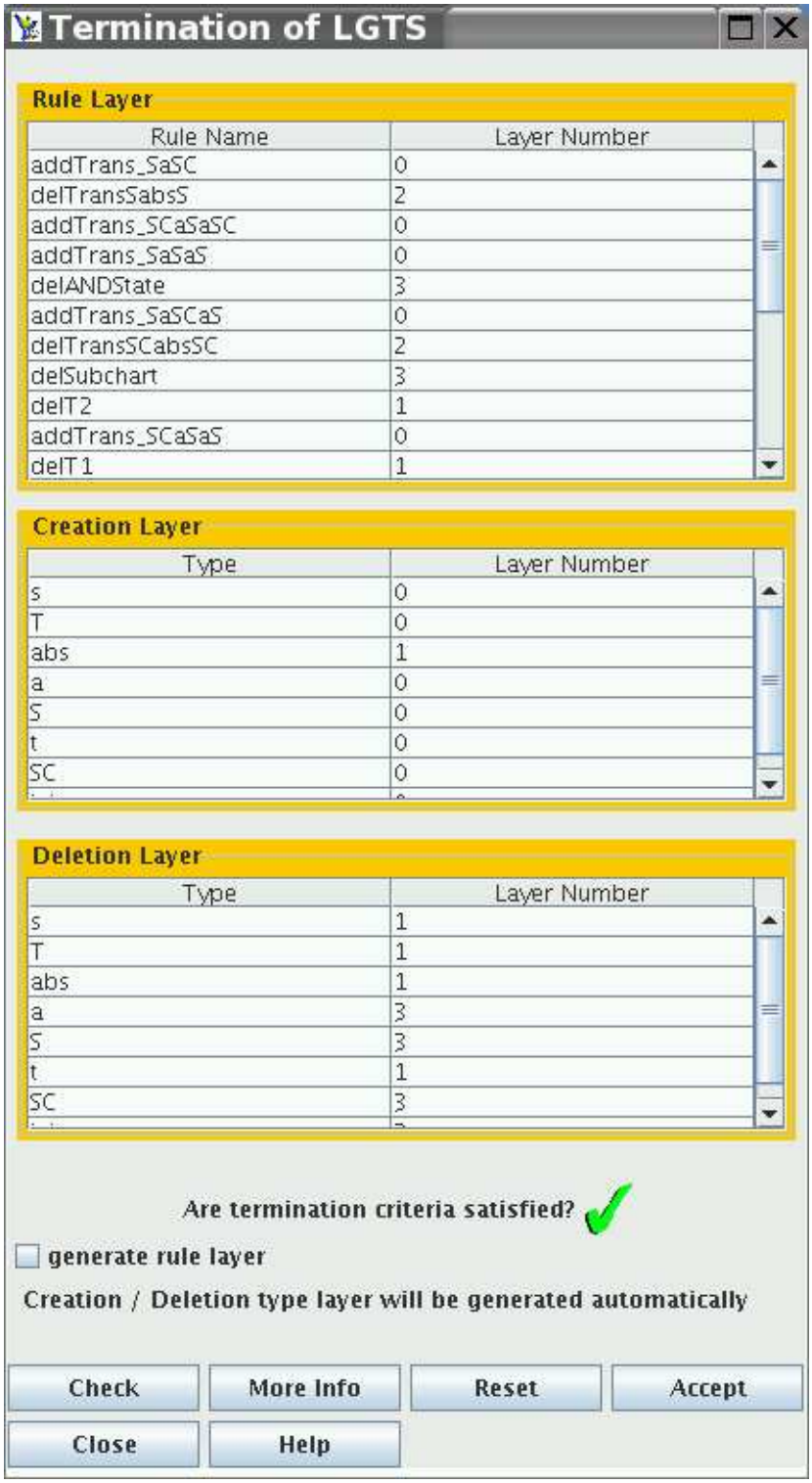


Figure 57: Termination dialog

A *layered graph grammar* with deletion and nondeletion layers terminates, if for each layer the deletion and nondeletion layer conditions defined above are satisfied.

Termination criteria are described in:

- "Termination Criteria for Model Transformation" by H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varro, and S. Varro-Gyapay (`http://www.cs.tu-berlin.de/~ehrig/public/EEL+05.pdf`).
- "Termination of High-Level Replacement Units with Application to Model Transformation" by P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer (`http://tfs.cs.tu-berlin.de/agg/BotKocParTaeENTCS.pdf`).

4 Help

AGG menu Help provides the following items:

- About AGG - Information about the AGG version and license terms. A brief description of AGG visual language.
- Menu Guide - A brief description of AGG's menus.
- Type Editor - A brief description of the Type Editor menu.
- Failures - Some typical matching and transformation failures and some tips for repairing them.