DiaGen

The Tree Tutorial

© University of Erlangen
http://www2.informatik.uni-erlangen.de/DiaGen/
diagen@immd2.informatik.uni-erlangen.de

October 6, 2003

Contents

1 Steps for writing a *DiaGen* editor **5**

2 Declaring components and relations **6**

3 Writing component and relation implementations **9**

3.1 Inheriting from pre-defined component classes 9
3.2 Attachment areas 10
3.3 A factory for creating components 10
3.4 Spatial relations 11
3.5 Testing the components 12

4 Defining the hypergraph analysis **14**

4.1 Reducer productions 14
4.2 Optimizing the production matching 15
4.3 Symbols of the hypergraph grammar 17
4.4 A grammar for tree diagrams 17
4.5 Testing the syntactic analysis 19
4.6 Optimizing the parsing process 20
4.7 The gluing condition 20
4.8 Geometric application conditions 22
4.9 General guidelines for writing the reducer and parser 23

5 Defining constraints for the diagram layout **24**

5.1 Constraints in reducer productions 24
5.2 The effect of constraints in the editing process 25
5.3 Geometric attributes for parser symbols 26
5.4 Constraints in parser productions 27
5.5 Trying out the constraint effects 29

6 Specifying complex editing operations **30**

6.1 A sample operation 30

6.2 Operator rules 31

6.3 Testing the operation 33

6.4 Creating relation edges in operations 33

6.5 Creating new components in operations 35

6.6 Initializing new components 36

7 Where to go from here **38**

Abstract

The following tutorial shows how a simple tree editor is constructed in the *DiaGen* framework. The tutorial assumes that you have read the *general introduction* (as PDF or HTML) to the system and are familiar with the *DiaGen* architecture and some basic terminology.

1 Steps for writing a *DiaGen* editor

Specifying a diagram type (and a corresponding editor) in *DiaGen* requires you to describe four different aspects:

- the visual appearance of the diagram, i.e. the visible diagram components and the spatial relations between them that are important
- the logical diagram structure, which is described by hypergraph transformation rules and a hypergraph grammar
- constraints on the diagram layout, which help to maintain this structure
- syntax-directed editing operations (similar to macros), which provide an easy way to implement complex manipulations of the diagram that are needed frequently

We suggest that you develop a diagram type specification in the following order

1. Define the formal syntax for the diagrams (the structure of the SRHG): determine what components you will need and how they will be connected
2. Write the Java classes that represent the diagram components (or reuse/adapt existing components) and code the spatial relation predicates.
3. Specify the reducing transformations and the hypergraph grammar to analyze a SRHG representation of a diagram

   This step is largely independent from step 2, but writing the Java code first allows you to create a executable test editor to try out your grammar
4. Define appropriate constraints for recognized structural relationships that preserve them when the diagram is modified
5. Define complex editing operations as transformations of the SRHG; make sure that the constraints enforce a correct layout for the transformed diagram

In this tutorial you will develop an editor for simple rooted tree diagrams. You will need Java version 1.2 or higher and, of course, the *DiaGen* framework classes that are contained in the “diagen.jar” file. Make sure the jar-archive is contained in your Java class path. To set up the environment, create a working directory “tree” and execute all commands from there. The ftp code directory for the tutorial contains all the source code file that you will be asked to create in the tutorial, so you can always copy them from there instead of typing them in yourself.

2 Declaring components and relations

Our tree diagrams should contain nodes and edges between them. We want the nodes to be represented as circles and the edges as arrows. The only spatial relation that is important for our diagram type is an "inside" relation; this relation should describe that the endpoint of an arrow lies inside a circle. Spatial relations between circles are not relevant for describing the structure of tree diagrams (although we could introduce them, if we wanted to require that nodes in a correct diagram do not overlap, for example).

Consequently, a circle needs one single attachment area to connect with arrows (its entire interior) while the arrows need two attachment areas, one for each endpoint. We choose to label both arrow endpoints identically because they both can be related to a circle area in exactly the same way.

Figures 1 and 2 show a sample tree diagram and the formal syntax representation that we want to be created from it.

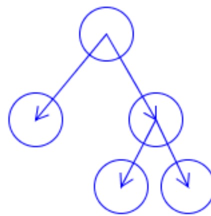

Figure 1: A tree drawing

Now you can write down the component and relation definitions in the *DiaGen* specification syntax. Create a file "spec" (the name is arbitrary) in the working directory with the following content.

[File from code directory: tree1.spec]

```
component circle[1] { Circle[CircleArea] },
          arrow[2] { Arrow[ArrowEnd,ArrowEnd] };

relation inside[2] { Inside[ArrowEnd,CircleArea] };
```

We augment this specification with empty declarations for the reducing transformations and hypergraph grammar. To make the specification valid, we must

Figure 2: The SRHG for the drawing in Fig. 1

include the definition of a starting symbol for the grammar; don't care about that now, we will come back to defining a grammar later.

```
nonterminal Tree[1];

reducer { }
grammar { start Tree; }
```

Now you can already run the generator (the class `diagen.generator.Generator`) on the incomplete specification file to create the class skeletons for components and relations: `java diagen.generator.Generator -f -s -p spec`

The -f flag tells the generator to overwrite existing source files, the -s flag causes the class skeletons to be created and the -p flag is necessary to turn off some optimizations, which would otherwise optimize the empty grammar away and cause run-time errors. If the specification file contains no errors, the generator output should look like this:

```
WARNING: Start symbol Tree is not reducible.
WARNING: The following symbols cannot be reduced to terminals: [Tree]
Writing Inside.java
Writing Circle.java
Writing Arrow.java
```

The two warnings are caused by the fact that we can never create correct drawings with respect to the empty grammar, but for now you can just ignore them. After

you have run the generator, you will find several new Java source files in your working directory:

The “Transformer.java” and “Grammar.java” files contain code for the reducer and parser, respectively. You should be able to compile them right away with javac.

The files “Circle.java“, “Arrow.java” and “Inside.java” contain class skeletons for the components and relations you have defined. Those files must be hand-edited before they can be compiled.

3 Writing component and relation implementations

Implementing a component class means that you have to code several aspects:

- the component geometry: the underlying parameters and the visual representation that is displayed in the editor
- the information that is needed to build the SRHG: the hypergraph edge label and the attachment areas
- means of manipulating the component: handles and possibly component-specific editing actions (property dialogs etc.)

Circles and arrows are among the pre-defined component types that are provided by the *DiaGen* library, therefore you do not need to write code for their visual representation and manipulation. The component classes in your working directory can simply inherit the standard behavior.

3.1 Inheriting from pre-defined component classes

Edit the class definitions in Circle.java and Arrow.java, remove the comments that suggest extension code and change the "`extends StandardComponent`" declarations to "`extends diagen.editor.lib.Circle`" and "`extends diagen.editor.lib.Arrow`", respectively.

Instead of editing the classes yourself, you can find the modified source files Circle.java, Arrow.java and Inside.java in the code directory

In order to understand the following definitions, you need to know the parameters that define the position and shape of the base classes:

A circle is defined by its center (the parameters `xcenter` and `ycenter` , which are combined in the `center` variable) and the radius parameter. An arrow is defined by its starting and ending points start and end, which are based on the `xstart/ystart` and `xend/yend` parameters respectively.

The constructors of both classes must be changed to pass on initial parameter values to the superclass constructors:

[File from code directory: Circle.java]

```
Circle(double x, double y, double r) {
  super(x,y,r);
}
```

[File from code directory: Arrow.java]

```
public Arrow(double x1, double y1, double x2, double y2) {
  super(x1,y1,x2,y2);
}
```

3.2 Attachment areas

The only part of the components that you have to define by hand is the creation of the attachment areas, which is done in the `initAttachs` method. For the circle area, you can use the library class `ShapeArea`, which defines an attachment area that includes the whole area of a component's shape (resp. its bounding box). Change the dummy definition in the `initAttachs` method of the Circle to:

[File from code directory: Circle.java]

```
protected void initAttachs() {
  attachs[0] = new ShapeArea(ATTACH_NAMES[0], this);
}
```

You should always use the values from the ATTACH_NAMES constant array to make sure the definitions in your class source correspond to the assumptions of the *DiaGen* generator.

For the arrow endpoints, you can use the `PointArea` class. This class implements an attachment area whose extent is defined as a small square around a given point. Consequently, this point must be defined in the constructor call for the attachment area.

[File from code directory: Arrow.java]

```
protected void initAttachs() {
  attachs[0] = new PointArea(ATTACH_NAMES[0], this, start);
  attachs[1] = new PointArea(ATTACH_NAMES[1], this, end);
}
```

3.3 A factory for creating components

Finally, we must provide a way for the editor to create new components by implementing factory objects. The code skeletons already contain a template for a default factory and you just need to fill in the constructor calls to give the initial parameter values.

For the circle, we place the center at the current cursor position and set the default radius to 20:

[File from code directory: Circle.java]

```
public static final double DEFAULT_RADIUS = 20.0;

public static final ComponentFactory FACTORY = new ComponentFactory() {
  public DiagramComponent create(Model model, Point2D[] pos,
                                Defaults defaults) {
    return new Circle(pos[0].getX(), pos[0].getY(), DEFAULT_RADIUS);
  }
};
```

For the arrow, we let the user specify two points on the drawing plane, which are then used as the start and end points:

[File from code directory: Arrow.java]

```
public static final String[] PROMPTS =
{ "Arrow start", "Arrow end" };

public static final ComponentFactory FACTORY
    = new ComponentFactory(PROMPTS) {
    public DiagramComponent create(Model model, Point2D[] pos,
                                  Defaults defaults) {
    return new Arrow(pos[0].getX(), pos[0].getY(),
                     pos[1].getX(), pos[1].getY());
  }
};
```

That completes the component definitions; the library superclass takes care of the rest. If you want to know more about writing your own component classes, you can find more information about coding the remaining aspects of a component in the *component Tutorial* (as PDF or HTML).

3.4 Spatial relations

Now we must define the relation test predicate for the `Inside` relation class by filling in the body of the `test` method. The mathematical formula for testing whether a point lies inside a given circle is, of course

```
  dist(p, center) <= radius
```

To code this test in java, we have to access the necessary parameter values from the corresponding attachment areas. When the `test(a1,a2)` method is invoked, we already know that `a1` refers to an attachment area with the "`ArrowEnd`" label and `a2` to one that is labeled "`CircleArea`". Therefore the component that `a2` belongs to must be of the "`Circle`" class and we can access the parameters through it:

[File from code directory: Inside.java]

```
protected boolean test(AttachmentArea a1, AttachmentArea a2) {
  Circle c = (Circle) a2.component;
```

This is the normal way to retrieve parameter values in a relation test, but for the arrow endpoints we need to use a little coding "trick" because we do not know whether it refers to the start or the endpoint of the arrow.

We could use different attachment labels "`ArrowStart`" and "`ArrowEnd`" to distinguish them; in this case we would need to define two relations that map to the same relationship edge label.

Instead, we do not access the point position through the component but we get it directly from the attachment area:

```
  java.awt.geom.Point2D p = ((diagen.editor.lib.PointArea)a1).getPoint();
```

Now we have all the values available that we need for the relation test:

```
  return p.distance(c.getCenter()) < c.getRadius().getVal();
```

Because the relation test is coded directly in Java, you can use all features of the Java language for it, as well as any available library methods (as `Point2D.distance` in our case). Don't forget that you are dealing with floating point values and that diagrams are usually only approximately correct. Therefore you should never test for exact equality but always include an error margin. The `SpatialRelation` class provides several constants and static methods to facilitate this.

3.5 Testing the components

Now you should be able to successfully compile all Java source files in the working directory by executing `javac *.java`

As soon as you have done this, you can try a prototype of your tree editor. To do this, run `java diagen.editor.ui.Editor DiagramType`. You should see

an editor window as shown in Fig. 3. `DiagramType` is a class which has been created by the generator; this class completely describes – together with the other generated classes – the tree diagram language and the customization of the generic *DiaGen* editor `diagen.editor.ui.Editor`. You can always test your generated editors by running this generic editor with the generated DiagramType class as parameter.

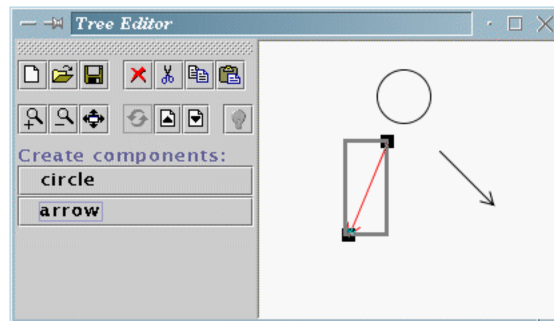

Figure 3: Screenshot of the tree editor prototype

You can use the right mouse button or the toolbar to invoke various commands, most notably, to create circle and arrow components. If you use the toolbar to create components, they will appear at the last position that you have clicked at on the drawing canvas. When you select a component it will display "handles" (small black boxes and a gray frame); you can alter a component's shape by dragging those handles. To select multiple components, you can click on an empty spot on the drawing canvas and then drag a selection frame across several objects, or you can select them one by one by holding the control button while you click on them.

The current user-interface is quite primitive, and we plan to refine it in the future. We also want to make the editor canvas available as a Java Bean so you can easily include it in your own application frames and write your own controls for it.

You can check your component and spatial relationship specifications by comparing your drawings with the SRHG which is created by the editor. To visualize the SRHG, invoke your editor with the property "`gml`", i.e., `java -Dgml diagen.editor.ui.Editor DiagramType` in our example. The editor will then write a file `srhg.gml` into your current directory and overwrite it after *each* editing operation. This file contains the SRHG which can be visualized with *Graphlet*, a general graph edit and layout tool from the University of Passau (`http://www.fmi.uni-passau.de/Graphlet`). Actually, all the hypergraph representations in this tutorial have been produced this way. Of course, their layout has been adjusted manually as you will have to if you use *Graphlet*, too.

4 Defining the hypergraph analysis

Next, we need to specify the logical structure of tree diagrams so they can be analyzed and tested for correctness. Diagram analysis is a two-step process: In the reducing step, a graph-transformation converts the SRHG into a simpler representation (the HGM). The HGM is then analyzed by a hypergraph parser. If you look at the edge types in the following productions, you can see that the reducer productions transform component and relationship edges into terminal edges, while the parser productions translate terminal and nonterminal edges into other nonterminal edges.

4.1 Reducer productions

Reducer productions translate meaningful patterns in the SRHG to simpler patterns in the HGM. In our tree diagrams, there are two patterns that are of interest: First, any circle corresponds directly to a tree node. Second, we need to detect edges between circles, i.e. a pattern of an arrow and two circles that are linked by inside relations.

The entire refined version of the specification file that is explained in this section can be found in the code directory. Of course, this is not the only way to describe the structure of trees; other reducing/parsing rules are possible that might even allow for a little more efficient parsing. For example, the reducer productions could already distinguish root, internal and leaf nodes by the use of negative context specifications.

Before we can write the reducer productions, we must declare the terminal edges that will make up the resulting HGM. We will create two types of terminal edges: nodes (those “nodes” in the tree are represented as hyperedges in the HGM!) and connections between them, which are represented as “child” hyperedges. Add the following line to the spec file after the relation declaration:

[File from code directory: tree2.spec]

```
terminal tNode[1], tChild[2];
```

The number in brackets defines the arity of the hyperedges (the number of nodes that an edge connects to).

The productions for the two patterns go inside the “`reducer ...` ” section: A reducer production consists of a SRHG pattern, an arrow “==>” and a HGM

pattern (and other optional parts); they are terminated with a semicolon. The two productions for the tree reducer are thus written as

```
circle(a) ==> tNode(a);

arrow(b,c) inside(b,a) inside(c,d) circle(a) circle(d) ==> tChild(a,d);
```

Arbitrary identifiers can be given for the nodes; using the same identifier again inside the scope of a production defines the connections between hyperedges.

4.2 Optimizing the production matching

Note that, although we are dealing with arbitrary graphs here, the sequence of edges in the SRHG pattern does matter, because it determines the order in which the reducer will look for the edges. You should always arrange your patterns so as to minimize the probability that a partial match cannot be extended. Typically you should go along a path (resp. a spanning tree) in the graph pattern where every edge has at least one node in common with edges that have already been matched.

For example, if you would write the pattern for the second rule as

```
arrow(b,c) circle(a) circle(d) inside(b,a) inside(c,d)
```

this would also work, but the reducer would start to match any arrow with any pair of circles before it detects that they are not connected and the partial match is aborted.

In the reducing step, all the reducer productions are applied to all possible matches in the SRHG and their right-hand sides are combined to form the HGM. (This process is deterministic, because the productions do not modify the SRHG). Figures 4, 5, and 6 show an example of a SRHG (note that the diagram does not represent a correct tree!) and the HGM that is produced by the reducing step.

You can check your reducer specifications by comparing your drawings with the HGMs which are created by the editor. If you specify the “`gml`” property (see page 13), the editor writes not only its SRHG to file `srhg.gml`, but also the HGM into a file `reduced.gml` in your current directory. Again, you can use *Graphlet* to visualize the HGM which is represented by this file.

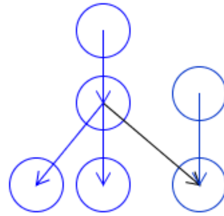

Figure 4: An incorrect tree drawing

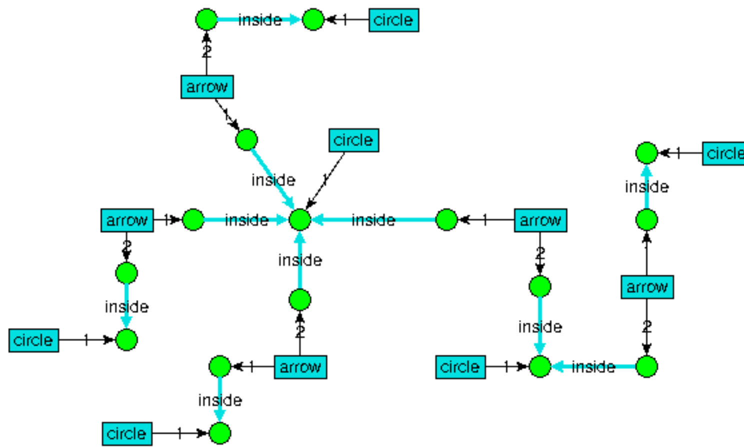

Figure 5: The SRHG for the drawing in Fig. 4

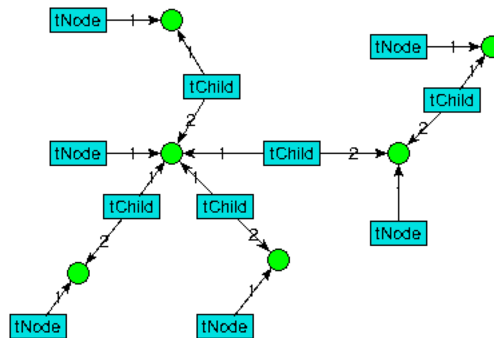

Figure 6: The HGM of the drawing in Fig. 4 produced by the reducer

4.3 Symbols of the hypergraph grammar

It is now easy to write a hypergraph grammar to parse the reduced graphs. We assume that you are familiar with standard context-free string grammars, which are used frequently in computer science. The hypergraph grammars that are used in *DiaGen* follow the same principles; the main difference is that the RHS of a grammar production does not specify a linear sequence of symbols but a graph of several hyperedges whose structure is defined by their shared nodes.

As a consequence, there are generally a lot more possibilities to apply inverse productions and derive new nonterminals from those that have already been found. This means that you have to be careful about specifying your grammars; otherwise you can easily end up with exponential (or even worse) parsing complexity.

Before we write the grammar itself, we again need to declare all the nonterminal hyperedges that will be constructed by the parser productions. We have already declared a "`Tree`" nonterminal which will represent an entire correct tree. In addition, we will need a " `Subtrees`" nonterminal, which we will use to accumulate all the child trees of a node before we combine them with the parent node to form a complete tree. Figure 8 shows how a part of a HGM should be analyzed by the parser.

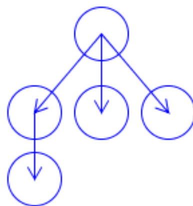

Figure 7: Another (correct) tree diagram

Now add the declaration of the "`Subtrees`" nonterminal:

[File from code directory: tree2.spec]

```
nonterminal Tree[1], Subtrees[2];
```

4.4 A grammar for tree diagrams

In the "grammar" section, we have already declared Tree as the starting symbol, i.e. the nonterminal that represents a correct diagram. Now we add grammar pro-

Figure 8: Syntax tree structure for the corresponding HGM of the drawing in Fig. 7

ductions to describe how detected structures are combined. A grammar production consists of a single LHS nonterminal, the definition sign "::=" and a sequence of (usually connected) hyperedges on the RHS.

A valid Tree can simply consist of a single node; alternatively, it can combine a node and the Subtree nonterminal that represents all child trees and the edges that link the node to them:

[File from code directory: tree2.spec]

```
Tree(a) ::= tNode(a);
Tree(a) ::= tNode(a) Subtrees(a,b);
```

A Subtree is formed by a Tree together with an incoming edge or by adding such a combination to another Subtree with the same root:

```
Subtrees(a,b) ::= tChild(a,b) Tree(b);
Subtrees(a,b) ::= [ [ tChild(a,b) Tree(b) ] Subtrees(a,c) ];
```

The purpose of the brackets in the second alternative is to specify the order in which the RHS parts are to be combined by the parser; in this case, it looks for a pair of `tChild` and `Tree` Symbols first and later joins them with another Subtree. If you do not group production RHSs that consist of more than two symbols explicitly, the default is a left-associative grouping.

In some cases, this ordering may affect performance. The generator transforms the context free part of the hypergraph grammar into extended CNF, which allows only productions of the form

NT ::= T
NT ::= NT
NT ::= NT NT

This means that no more than two symbols can be combined in one inverse production application; if necessary, new nonterminals and internal productions are added to the grammar to achieve this.

Note that the grammar is context-free, i.e. the left-hand side of the productions consists of a single nonterminal. Because this formalism is not always expressive enough to describe a specific diagram language, the *DiaGen* parser also allows another type of productions, which we call “embedding” productions. To find out more about the capabilities of the diagram analysis module and the constructs that you can use in reducer and parser rules, have a look at the *DiaGen specification grammar*.

4.5 Testing the syntactic analysis

You can now regenerate the “Transformer.java” and “Grammar.java” files by running the generator again: `java diagen.generator.Generator -f spec`

Omit the -s and -p flags this time, because you do not need to regenerate the skeletons – the component and relation classes are already finished – and the grammar should be correct now as well. You should see some output like this:

```
Overwriting existing DiagramType.java.
Overwriting existing Grammar.java.
Overwriting existing Transformer.java.
```

The warnings have gone now, indicating that the generator could not find anything suspicious about the new grammar.

Compile the modified Java files and run the prototype again. You will notice at once that all the circles are now colored in blue-green shades: this is how *DiaGen* editors indicate correct diagram parts. According to our grammar, every circle is by itself a correct tree of height 0. Try to draw some larger diagrams and see how the coloring indicates the substructures that the parser could reduce to the Tree symbol.

If you use property “`gml`” (see page 13), you can check the derivation structure of each of the detected correct subdiagrams, i.e., trees. Property “`gml`” triggers the editor to write files `subdiag-`n`.gml` into your current directory ($n = 0, 1, 2, \ldots$),

a single one for each of the detected subdiagrams. For our tree example, each file contains the derivation tree of the corresponding sub-HGM which can be visualized by *Graphlet*.

4.6 Optimizing the parsing process

While context-free string grammars can always parse any string in polynomial O(N^3) time, this is unfortunately not true for hypergraph grammars. If you think of a large tree structure, *any* connected subgraph that has no dangling edges also forms a tree. The hypergraph parser is required to find not just one way of parsing the diagram correctly, but it must find *all* possibilities and *all* correct subdiagrams. For our grammar, this leads to an exponential explosion in parsing time.

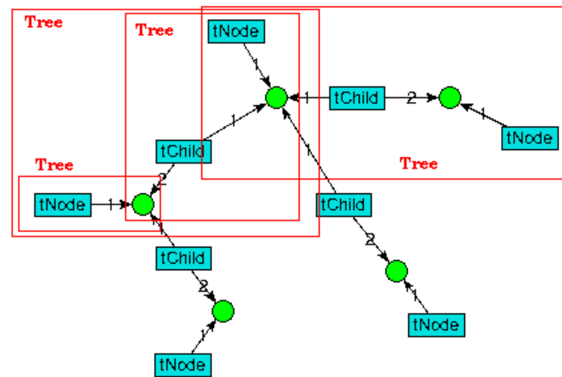

Figure 9: Some extraneous partial trees in the HGM from section 4.4

TODO: mention debugging options to check reducing and parsing, show how the problem can be found

To avoid this, we must force the parser to detect only those subdiagrams that are actually needed in the parse tree for the whole diagram (or just “a few” more). That means we want to proceed “bottom up” from the leaves to the root of the tree and we want to ignore “tree” structures that still have arrows leading out of their “leaves”.

4.7 The gluing condition

The usual way to prevent the parser from detecting unwanted substructures is to add application conditions to the productions. In our case, one possibility would

be, to detect leaf nodes in the reducer (they have no outgoing arrows) and to start parsing from there. Instead of rewriting the whole specification to do this, we will use a simple shortcut here, which has almost the same effect: We can enforce a "gluing" application condition for the second (recursive) Subtree production by appending the "!" operator to the production:

[File from code directory: tree3.spec]

```
Subtrees(a,b) ::= tChild(a,b) Tree(b);
Subtrees(a,b) ::= [ [ tChild(a,b) Tree(b) ] Subtrees(a,c) ] ! ;
```

The effect of the gluing condition is, informally stated, that the corresponding production can only be applied if all the connections that are "lost" by applying the inverse production have already been included in the parse tree.

A more formal description of the condition evaluation:

- Take $S_{\text{unreachable}}$, the set of all nodes that appear on the RHS of the production, but not on the LHS.
- Find S_{lost} , the set of all terminal edges, which connect to those nodes.
- Construct S_{parsed} , the set of all terminals that can be derived from the RHS nonterminals (and thus also from the LHS, if the inverse production is applied)
- The inverse production can only be applied, if S_{lost} is a subset of S_{parsed}.

Using gluing condition entails no performance penalty, because the S_{parsed} sets are always passed along for every nonterminal that the parser finds; they are needed to ensure that the extensions of the RHS symbols in a production are disjoint.

In our production

```
Subtrees(a,b) ::= tChild(a,b) Tree(b) Subtrees(a,c)
```

the "c" node is not reachable from the new LHS terminal after the inverse production has been applied. Application is therefore prevented, if this node (which corresponds to the root of the last subtree that has been joined to the Subtrees(a,c) nonterminal) has any connections (i.e. outgoing edges) that are not part of its parse tree. Effectively, this enforces that a tree can only be incorporated into a higher-level tree, if it has been built up from leaves and it has no dangling edges pointing out.

Note that the RHS nodes usually have "unparsed" edges connected to the "root" node (a): these further subtrees at the same level and an arrow that leads into the root from a higher level. But this is perfectly admissible, as the a node is still present in the LHS and the connections can therefore be considered when further inverse productions are applied to the new nonterminal.

To ensure that a correct derivation is found, the parser should actually enforce the gluing condition for all inverse production applications. If the condition is violated, this would mean that you could not actually *derive* the diagram from the start symbol by applying the productions in the parse tree, because some nodes would have to appear "from out of nowhere" in that case. Nevertheless the parser does not check the gluing condition unless it is required explicitly, because that would make it impossible to partially parse many diagrams that are almost correct, except for some additional components.

4.8 Geometric application conditions

Thus we have now restricted the parsing process so that there are still some extraneous production applications, but they do not cause an exponential multiplication of subdiagrams any more. Unfortunately we still have a problem, because the grammar does not determine the order in which the branches are added to extend a `Subtree` construction and is thus ambiguous: When a node has two children, the grammar allows either to join the left edge and son to a `Subtree` symbol and then build a larger `Subtree` symbol by adding the right edge and son or vice versa. To analyze a diagram correctly, the parser is required to detect all partial parse trees that can be constructed from it; in our case it must apply the production again for every possible ordering of son nodes. Doing this recursively leads again to a exponential explosion in valid (sub)diagrams and thus in parsing time and space requirements.

Consequently we need to disambiguate the `Subtree` production in the grammar by enforcing a order among the sons. We can express this by adding an application condition:

[File from code directory: tree3.spec]

```
Subtrees(a,b) ::= tChild(a,b) Tree(b);
Subtrees(a,b) ::= [ [ tChild(a,b) Tree(b) ] Subtrees(a,c) ] !
                  if b.x < c.x || b.x == c.x && b.y < c.y ;
```

The condition compares the positions of the nodes that represent the root of the child tree that is about to be added to the `Subtree` (the b node) to that which has been added before (the c node): Every node in the SRHG and HGM has x

and y attributes associated with it that represent the center of the corresponding attachment area and those can be used for geometric application conditions. In our case we require that the new child tree is to the left of the last one, so the child trees are combined from right to left.

To avoid any ambiguity, we also specify an arbitrary ordering if the nodes should happen to lie on the same x coordinate. A tree will not be parsed correctly if two son nodes should lie exactly on top of each other, but we can assume that this does not occur in a correct drawing. Note that, if a node has n sons, there are still O(n^2) ways to build partial combinations of them, but only one that includes all the nodes. Because of the gluing condition introduced earlier, we have thus avoided the exponential propagation of parsing possibilities across tree levels.

You can rerun the generator at this stage and compile and run the editor prototype, to make sure your changes are correct. You won't see many changes in the behavior of the editor though, unless you create trees with more than a few nodes, where the exponential time requirements of the parser would have come into play with the unoptimized grammar.

4.9 General guidelines for writing the reducer and parser

When you define a grammar for a diagram language, you should try to design it in such a way that it can also analyze incorrect diagrams as far as possible. This goal may sometimes conflict with the desire to get optimal parsing efficiency for correct diagrams; for example, the application conditions that we have just introduced, will prevent the parser from recognizing a tree structure in the diagram if there exists a separate node with a link pointing *into* one of the child nodes. Therefore we recommend that you introduce restrictive application conditions only to avoid unacceptable performance penalties (i.e. exponential parsing complexity). You should rather use a simple and general grammar than try to optimize the parsing process as much as possible.

Both the reducer and the parser use hypergraph productions to analyze the diagram. The reducer rules allow more powerful constructs (in particular, they are not restricted to be context-free or embedding productions), but they are applied only once to produce a new hypergraph model. The parser productions, on the other hand, are more restricted, but they are applied recursively to build hierarchical structures. As a consequence you should use the reducer to simplify the abstract diagram representation as much as possible before you detect the nesting structure of the diagram with the parser. This separation is a bit similar to that of lexical and syntactic analysis in traditional compiler construction.

5 Defining constraints for the diagram layout

The analysis part of the tree editor now works as expected, but it is completely separated from the visual editing process. This situation is similar to a classical edit-compile cycle in program development, where the editor and the compiler are separate programs that do not know about each other. For textual programs this is generally an acceptable situation, but we prefer to have features like automatic indentation and syntax highlighting, which require some syntactic analysis of the source code. A user-friendly graphical editor must provide even more support because laying out graphics correctly by hand requires more work than writing syntactically correct programming code.

Automatic layout can be integrated into a *DiaGen* specification by defining constraints on the geometric attributes of the diagram components. When *DiaGen* operates in its "intelligent mode", a constraint solving engine is activated that recalculates the layout according to these constraints while the user changes a diagram. The constraints thus propagate local changes and adjust other diagram parts to maintain the (partial) correctness of the diagram.

We have not developed our own constraint solving engine for *DiaGen*, but instead we rely on existing constraint solvers that are integrated into the system with the help of some adapter classes. *DiaGen* can currently interface with the *Qoca* constraint solver from Monash University, Australia and the *ParCon* constraint solver from the University of Paderborn.

5.1 Constraints in reducer productions

For the tree editor example, we will use *Qoca* , because it is completely written in Java and runs on any platform that *DiaGen* can be used on. *Qoca* is freely available under GPL and it is included in the diagen.jar file, so you do not need to install it separately. *Qoca* supports linear equality and inequality constraints, which is all we need for this example.

The code that provides the necessary interface is contained in the class diagen.editor.param.QocaLayoutConstraintMgr, therefore you need to include the following line in the specification file before the reducer declarations:

[File from code directory: tree4.spec]

```
constraintmanager diagen.editor.param.QocaLayoutConstraintMgr;
```

The most obvious constraint that will help for editing tree diagrams, is to link the arrow ends to the node positions. Such a connection is recognized by the second reducer production, so we add constraint specification to it that fixes the arrow endpoints (defined by the arrow's `xStart/yStart` and `xEnd/yEnd` parameters) at the center of the circles (defined by their `xAnchor/yAnchor` parameters) You must also assign symbolic names to those production edges (components) that you want to refer to in the constraints:

```
ar:arrow(b,c) inside(b,a) inside(c,d) c1:circle(a) c2:circle(d)
==>
tChild(a,d) {
   constraints: ar.xStart == c1.xAnchor; ar.yStart == c1.yAnchor;
                ar.xEnd == c2.xAnchor;   ar.yEnd == c2.yAnchor;
};
```

With the current *DiaGen* release, you have to look in the component source files to find out the parameter names. For the future, we plan to include the parameter definition in the specification.

As you can see, every reducer or grammar production in a *DiaGen* specification can have associated constraint specifications (and also computations of user-defined attributes). The "constraints" keyword separates the constraints from other actions (user-defined attribute computations). All those actions are given in braces at the end of the production and they are evaluated whenever the corresponding (inverse) production is applied. To be exact, actions that belong to reducer productions are executed for *all* the matches (and thus applications) of the production in the SRHG. Actions associated with parser productions, on the other hand, are executed (after the paring is complete) only for those productions that actually derive the drawing from the starting symbol, i.e. those that are part of the valid parse tree that the parser has found for the diagram. Actions for unneeded (dead-end) productions are not executed. In incorrect diagrams, parse actions are executed for the productions that form the detected partial syntax trees.

5.2 The effect of constraints in the editing process

You can already run the generator, compile the java sources and try out the effect of the changes: First of all, you will see, that the editor now comes up with "intelligent" mode turned on by default, because the diagram type now has a valid constraint manager. As soon as you draw a construction that matches the second reducer rule (two circles connected by an arrow), you will notice that the arrow ends "snap" to the center points of the circles and remain there if you move the

circles around. To disconnect the arrow again, you temporarily have to turn off the intelligent editing mode (press the toolbar button with the light bulb).

5.3 Geometric attributes for parser symbols

Now we can add more constraints that will provide a more restrictive tree layout. To do this, we have to attach constraints to grammar productions as well, but the grammar productions can only access terminal and nonterminal edges and thus cannot set component parameters directly. Therefore we need to introduce additional attributes for the terminal and nonterminal edges, which can then be indirectly linked (by constraints) to the component parameters that define the actual diagram layout. Attribute definitions for parser edges must be included in braces after the edge declaration:

[File from code directory: tree5.spec]

```
terminal tNode[1] { Variable xpos, ypos; },
         tChild[2];

nonterminal Tree[1] { Variable xroot, yroot, xleft, xright; },
            Subtrees[2] { Variable xrleft, xrright, xleft, xright, ytop; };
```

The intended meaning for the attributes is as follows:

tNode	xpos/ypos	the node center
Tree	xroot/yroot xleft/xright	the root node position the x coordinates of the leftmost and rightmost leaves
Subtrees	xrleft/xrright xleft/xright ytop	the x coordinates of the leftmost and rightmost roots of the subtrees the x coordinates of the leftmost and rightmost leaves coordinate of the subtree roots (we want them all to be aligned on the same level

Next, we add constraints to the first reducer production to set the node position equal to the center of the corresponding circle:

```
c:circle(a) ==> t:tNode(a) {
  constraints: t.xpos == c.xAnchor; t.ypos == c.yAnchor;
};
```

The equality constraints that pass on the attribute values are specially optimized in the *DiaGen* implementation; all the variables are internally made to refer to the same value, so there is no actual "constraint solving" involved.

5.4 Constraints in parser productions

Then we move on to the parser productions, first those for the Tree symbol. If the tree consists of a single leaf node, then its root, left and right coordinates are all equal to the position of the node.

[File from code directory: tree5.spec]

```
t:Tree(a) ::= n:tNode(a) {
    constraints: t.xroot == n.xpos; t.yroot == n.ypos;
                 t.xleft == n.xpos; t.xright == n.xpos;
  };
```

The attribute dependencies are more complex, if the tree is built from a Subtrees symbol that comprises one or more child trees. The root position is obviously that of the new root node:

```
t:Tree(a) ::= n:tNode(a) s:Subtrees(a,b) {
    constraints: t.xroot == n.xpos; t.yroot == n.ypos;
```

The leftmost and rightmost leaf positions are identical to those in the union of the child trees

```
                 t.xleft == s.xleft; t.xright == s.xright;
```

To enforce stricter layout rules, we impose additional constraints requiring that the root must always be horizontally positioned between the leftmost and rightmost child nodes.

```
                 s.xrleft <= t.xroot <= s.xrright;
```

For nodes with a single child, this means that the child and the parent node must be on a vertical line. We want our trees to grow downwards, so the child nodes must all lie below their root at a vertical distance of at least 50 units.

```
                 t.yroot <= s.ytop-50;
  };
```

Note that we are not defining application conditions here, but layout constraints. This means, that a correct tree will be *recognized* in a diagram even if it does not conform to the layout constraints, but when you turn on intelligent mode, it will be automatically reshaped to an admissible layout.

Finally, we must also add constraints to the Subtrees productions. If the Subtrees construction is built from a single child tree, we must simply set the leftmost/rightmost leaf positions from this tree

```
s:Subtrees(a,b) ::= tChild(a,b) t:Tree(b) {
    constraints: s.xleft == t.xleft; s.xright == t.xright;
```

and the leftmost and rightmost root are both identical to the root of the single child tree.

```
                 s.xrleft == t.xroot; s.xrright == t.xroot;
                 s.ytop == t.yroot;
  };
```

The most complicated case is that of combining a `Subtreess` construct with a new child tree to its left. The leftmost leaf is in the new child tree while the rightmost one comes from the old child trees; the same goes for the leftmost and rightmost roots:

```
s:Subtrees(a,b) ::= [ [ tChild(a,b) t:Tree(b) ] s2:Subtrees(a,c) ] !
  if b.x < c.x || b.x == c.x && b.y < c.y {
    constraints: s.xleft == t.xleft; s.xright == s2.xright;
                 s.xrleft == t.xroot; s.xrright == s2.xrright;
```

We want all the roots of child trees with the same parent to be aligned on the same y-coordinate:

```
                 s.ytop == t.yroot == s2.ytop;
```

Finally, the new subtree must not overlap the old ones horizontally, but its rightmost leaf must be positioned at least 50 units to the left of the leftmost leaf of the old Subtrees construct.

```
                 t.xright <= s2.xleft-50;
  };
```

5.5 Trying out the constraint effects

This completes our constraint specification. Create and run the editor prototype and see how the constraints affect the layout when you draw and modify diagrams. Our layout constraints are quite restrictive, so some diagram manipulations (like horizontally swapping two sibling nodes) are now prevented and you must turn the intelligent editing mode off to execute them.

6 Specifying complex editing operations

This last editor prototype is already a useful tool for creating tree diagrams; for this purpose, it is obviously superior to a general drawing program, because it recognizes and preserves the structure of the diagram. But there is more that you can do to support the editing process: Editing diagrams by manipulating the individual components is often a little tedious. For example, when you add a node to a tree, you have to create the new node and the connecting edge separately and you must reshape the edge so that it properly connects the new node to its parent, before the new structure is recognized and automatic layout can take place. It would be much nicer if you could do all that in just one step.

To simplify such heavily used editing operations, you can augment the diagram type specification with complex operations: Those combine several modifications of the diagram into one step, similar to "macros" in standard office programs, and try to use the constraint manager to lay out the modified diagram properly.

Complex operations are based on operator rules, which define basic transformations on the SRHG; they take a pattern of component and relationship edges and remove some of them or add new ones. Note that, unlike the reducer and grammar rules described above, the operator rules actually *modify* the SRHG, therefore the order in which several rules are applied is very important; changing it may lead to different results.

The operations themselves describe the sequencing or iteration of the rule applications. Normally the user must (partially) specify the context of the rule applications in the drawing.

6.1 A sample operation

We will start by introducing an operation that describes the deletion of a tree node along with all incoming and outgoing edges. The node will be specified by the user; the connecting edges can then be found automatically. First, you need to add an "operations" section to the end of the spec file:

[File from code directory: tree6.spec]

```
operations {
```

Now, inside that "operations" section goes the specification for the "delete" operation: Declare the operation name, the UI text string, which will appear in the

menu and an icon for the operation. (If you don't have an icon ready, you can use the default icon "images/plain.gif").Don't forget the colon that terminates the declaration.

```
operation delete "delete node" "images/plain.gif":
```

Then you describe, what components (component edges) the user must specify before the operation can be executed. For the delete operation, this is a single component of type "circle" (the SRHG edge label); you must give it an identifier, so you can refer to it later and you must also give a prompt string, that will be displayed while the user selects this component.

```
          specify circle c "node to delete"
```

Finally, the rule body specifies what operator rules must be applied, when the rule is executed: We need rules to remove the incoming edge, the outgoing edges and the node itself. The rule applications can be provided with a partial match for the rule pattern, which is given as a sequence of specified components; in our case, the selected node is always given as a partial match.

```
          do remove_outgoing(c)! remove_incoming(c)? remove_circle(c);
```

Special operators govern the rule application: The second rule is optional, because an incoming edge does not necessarily exist. The "?" operator specifies that the rule is simply ignored, if its pattern cannot be matched; for standard rule application, an error will be reported in this case. The "!" operator means that the removal of outgoing edges will be repeated as long as a match for the rule pattern can be found. When you use this operator, you must make sure that the same SRHG edges will not be matched again after the rule application, or else you will create an infinite loop where the rule is applied again and again to same context. You can do this either by removing pattern edges in the application or by including edges that are added by the application in a negative context for the rule. You can find more information about rule application operators in the *DiaGen specification grammar*.

6.2 Operator rules

To complete the operation definition, we must of course also define the operator rules that we have already used. The rule definitions must precede their use in the specification file; otherwise the generator will indicate an error. An operator rule

definition consists of a pattern of SRHG rules (as for a reducer rule) and a rule body that adds and deletes SRHG edges. For now, we only need to remove edges; to do this, repeat the edge identifier in the rule body preceded by a “-”.

[File from code directory: tree6.spec]

```
rule remove_circle:
     c:circle(a)
     do -c;

rule remove_incoming:
     circle(a) inside(b,a) ar:arrow(c,b)
     do -ar;

rule remove_outgoing:
     circle(a) inside(b,a) ar:arrow(b,c)
     do -ar;
```

When you run the generator on the spec file at this stage, it will work correctly but it will give you the following warnings:

```
WARNING: Line 70, column 14: relation edge inside(b,a) is removed
                                                     automatically
WARNING: Line 74, column 14: relation edge inside(b,a) is removed
                                                     automatically
```

As the SRHG model does not allow “dangling” relationship edges, removing a component edge automatically removes *all connected relationship edges* in the SRHG model. If any connected relationship edges are part of the rule pattern, you should also remove them explicitly in the rule body:

[File from code directory: tree7.spec]

```
rule remove_incoming:
     circle(a) r:inside(b,a) ar:arrow(c,b)
     do -r
        -ar;

rule remove_outgoing:
     circle(a) r:inside(b,a) ar:arrow(b,c)
     do -r
        -ar;
```

The generator will not create code for this, but it will know that you are aware of what you are doing. But keep in mind that connected relationship edges which are

not part of the pattern are silently removed, too. Of course you can also remove relationship edges on their own without modifying connected components.

Unlike the reducer and parser productions that we have seen so far, the operator rules actually *modify* the SRHG. Therefore you have to pay attention to the rule application sequence. When a non-optional rule cannot be applied (because the partial match could not be extended correctly), the whole operation is aborted and an error message is shown. However, the effects of all rules that have been executed up to this point remain visible. Therefore it is often a good idea to include a no-effect rule at the start of an operation, which just makes sure that all the required patterns are actually accessible.

6.3 Testing the operation

When you run the generator again, you will now notice that a new file "Operations.java" has been added to your working directory. Compile this file and the modified "DiagramType.java", run the editor again and try out the new operation. You can access it from the pop-up menu; operations also appear on the toolbar, if a component has been selected that is a suitable first argument. When you execute a complex operation, you will first be prompted to select the required components in the drawing. A pop-up dialog displays the prompt strings that you have given in the specification. While you are in the selection mode, you cannot access other editor operations; to abort the selection process and return to normal editing mode, press the "cancel" button in the dialog.

After the context for the operation has been selected, its body is executed and the rules modify the SRHG. The new SRHG is then parsed again and finally the resulting constraints are recomputed, hopefully producing a proper layout for the modified diagram.

6.4 Creating relation edges in operations

Finally, we will define operations for the tree editor that add edges to the SRHG. First, we will define an operation that merges two nodes; i.e. one of them is deleted and all its subtrees are moved to the other node. The operation specification looks like this:

[File from code directory: tree8.spec]

```
operation join "join nodes" "images/joinNodes.gif":
          specify circle c1 "node to delete",
                  circle c2 "node to move subtrees to"
          do check_path(c1,c2) move_outgoing(c1,c2)!
             remove_incoming(c1)? remove_circle(c1);
```

The rules that remove the node and the incoming edge have already been defined above. Moving the subtrees means that we delete the "inside" relation between the outgoing arrow and the first circle and instead add new "inside" relations connecting them with the second circle:

```
rule move_outgoing:
     circle(a) circle(d) inside(b,a) arrow(b,c)
     -{ inside(b,d) }
     do +inside(b,d);
```

In this case we use the negative application condition "`-{ inside(b,d) }`" to make sure the rule is not applied twice to the same outgoing edge. We could alternatively delete the inside(b,a) edge right away, but it will also be deleted automatically when the tree node circle(a) is removed later.

The `check_path` rule is a no-op rule that serves to ensure that the deleted node is not an (indirect) parent node of the join node. In that case the operation should not be applied because the resulting structure is not a valid tree any more.

```
rule check_path:
     circle(a) circle(b)
     if ! a -- ( inside(1,0) arrow(0,1) inside(0,1) )+ --> b;
```

As you can see, the rule describes no modifications (it has no "`do ...`" section), but it contains an application condition. The "`a -- ... --> b`" expression tests for a path between the a and b nodes:

The path description inside the arrow is constructed from edge sequences which consist of edge labels followed by two numbers. The numbers designate the links that lead the path "through" a hyperedge. Those edge sequences can then be used in (restricted) regular expressions; our example uses the "`( ... )+`" operator to express that the path may consist of 1 to n repetitions of the edge sequence "`inside(1,0) arrow(0,1) inside(0,1)`". Thus the entire path expression describes a path that leads from node a to node b through a sequence of child links; i.e. the path expression evaluates to true if circle(b) is a descendant if circle(a) and the negation operator "!" ensures that the rule cannot be applied in this case; this causes the entire operation to be aborted.

Figure 10 shows how two repetitions of the edge sequence map to a path from a node to a grandchild in the SRHG from section 2.

Figure 10: A path in the SRHG

Note that the links of a hyperedge are numbered from 1 in the figure, whereas link numbers path expressions start at 1. Link numbers for relation edges are not shown in the SRHG figures.

Remember that a rule application only changes the SRHG of the edited diagram; the spatial layout of the diagram is adjusted by the constraint system. Therefore the constraints in the diagram must be strong enough to arrange the diagram in a way that will actually reflect the relations that you have added. In our case, the new "inside" edge causes the arrow to be changed to its new origin, so the relation is in accordance with the geometric layout. It is considered to be an error, if the spatial relationships that have been introduced by an operator rule cannot be found again by the relation scanner in the new diagram layout resulting from the constraint computation. (In the current *DiaGen* implementation, this error is not reported, but the relations simply disappear again.)

6.5 Creating new components in operations

Adding component edges to a SRHG is a bit more complicated than adding relationships, because the corresponding component objects must be initialized correctly. There is no general way to do this, but instead the object creation is deferred to an external static Java method, which has to be coded by hand. As an example,

we will finally define an operation that takes a node as an argument and adds a branch (child node and edge) to it. The operation specification is straightforward:

[File from code directory: tree8.spec]

```
operation add "add node" "images/addNode.gif" :
          specify circle c "select parent"
          do add_child(c);
```

The `add_child` rule is responsible for creating the new SRHG edges. To create the component object corresponding to a new component edge, you must include the (fully qualified) name of a static method in braces after the edge creation statement. You can give component edges from the rule pattern as arguments for this method; they will be passed as arguments to the generated method call. The code for creating relationship edges can be produced by the generator.

```
rule add_child:
     cr:circle(a)
     do +circle(b) { OperationSupport.create_child_Node(cr) }
        +arrow(c,d) { OperationSupport.create_child_Arrow(cr) }
        +inside(c,a)
        +inside(d,b);
```

6.6 Initializing new components

To complete the definition, you must finally create the source file "OperationSupport.java" and define the component creators. The class can be declared "abstract", as it contains only static fields and no instances will ever be created for it.

[File from code directory: OperationSupport.java]

```
import diagen.editor.graph.*;
import diagen.hypergraph.*;

abstract class OperationSupport {
```

The `create_child_Node` method takes the parent node and creates a child 50 units below it. To access the component that correspond to the given edge, you have to cast it to a `ComponentEdge`. You can then call the `getComponent` method,

cast the result to a `Circle` and access the parameters.

```
static GraphComponent create_child_Node ( AppContext ctx,
                                          Edge parent ) {
    Circle c = (Circle) ((ComponentEdge) parent).getComponent();
    double x = c.getCenter().getX();
    double y = c.getCenter().getY();
    Circle nc = new Circle(x, y+50.0, 20.0);
    ctx.addComponent(nc);
    return nc;
}
```

Methods for component creation are always called with an `AppContext` object as a first argument in addition to the arguments that are given in the rule specification. This application context makes it possible to access user-controllable default settings like the standard font for text objects.

The `create_child_Arrow` method creates the connecting arrow in a similar way.

```
static GraphComponent create_child_Arrow ( AppContext ctx,
                                           Edge parent ) {
    Circle c = (Circle) ((ComponentEdge) parent).getComponent();
    double x = c.getCenter().getX();
    double y = c.getCenter().getY();
    Arrow na = new Arrow(x, y, x, y+50.0);
    ctx.addComponent(na);
    return na;
}
```

When you compile and try out this last editor version, you will notice that the new child nodes do not always appear at the position that is set in the creator methods: the constraint system prevents nodes from overlapping and moves them sideways if necessary. Knowing your constraint system, you must decide yourself, which parameter values for the new component must be set correctly and which can be set to default values, because they are recomputed by the constraint manager.

TODO explain construction of user defined attributes and semantics

7 Where to go from here

Congratulations, you have completed the tree editor tutorial! By now you should understand how the *DiaGen* system works, you have seen most of the constructs that you can use in a specification file and you should know how you can go about writing your own editors for more complex languages.

There are several places where you can look for further information:

The *DiaGen homepage* at the University of Erlangen contains more information including links to some conference papers that relate to *DiaGen*:

The *DiaGen Overview* (as PDF or HTML) describes some basic concepts and the general architecture of the system. You should already have read this before you started on this tutorial.

The *Component Tutorial* (as PDF or HTML) gives you more information about how to code your own diagram components in Java.

The *DiaGen specification grammar* shows you the admissible construct for specification files and gives brief explanations of what they do.

The *JavaDoc documentation* of the framework is also accessible on the Web

The *DiaGen* class tree contains several sample editors, some of which are considerably more complex than the simple tree example that has been described in this tutorial. These editors can be found in the subpackages of the diagen.editor.sample package.