Constraint-based Diagram Beautification

Sitt Sen Chok, Kim Marriott and Tom Paton
School of Computer Science and Software Engineering
Monash University, Victoria, Australia
{css,marriott,tomp}@csse.monash.edu.au

Abstract

Recently a new metaphor for diagramming, the intelligent diagram has emerged. This new metaphor provides an excellent basis for diagram beautification since it automatically collects geometric constraints which capture the semantics of a diagram during diagram construction. By applying visual-language-specific layout rules which impose additional desired constraints on the diagram component placement, we can provide powerful semantics preserving diagram beautification. We have demonstrated the feasibility of this approach by developing a simple extension to the Penguins system which generates beautification rules from a grammatical specification of the visual language and tested the system with three example visual languages: Binary trees, state transition diagrams and mathematical equations.

Keywords: Layout, Constraint, Graphic Editor, Visual Languages.

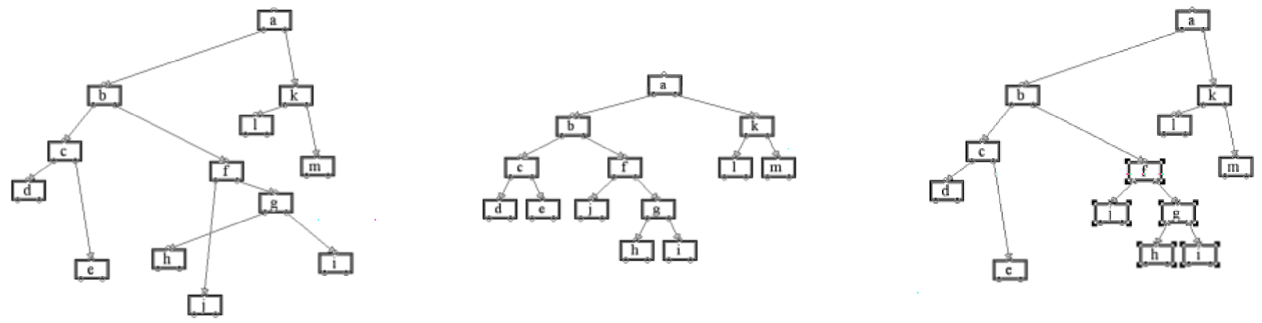
1. Introduction

When using diagramming tools it is common for people to first sketch the diagram with the tool and then to *beautify* the diagram where by beautification we mean fine-tuning the placement of objects and their appearance. Unfortunately, few diagram editors provide help with this second phase, in part, perhaps, for the following two reasons. First, the transformations used in beautification depend on the kind of diagram. For instance, rules for beautifying a tree are probably not appropriate for beautifying a state transition diagram and certainly not for beautifying a mathematical equation. Second, beautification should preserve the *semantics* of the diagram. For instance, beautifying a state transition diagram by making all transition arrows the same length will not be very useful if the arrows are no longer attached to the states. Because beautification relies on understanding the semantics and kind of the diagram, general purpose diagram editors have instead chosen to provide features such as snap-to-grid which help the diagrammer to immediately construct an aesthetically pleasing diagram. However, although these techniques are useful, there are many aspects to beautification, such as ensuring that nodes in a tree are aesthetically placed, which such tools cannot help with.

Fortunately, a new metaphor for diagramming, the *intelligent diagram* has emerged [1, 2, 3, 4]. This model combines the best features of uninterpreted diagramming and syntax-directed diagramming. In this model the graphic editor is specialized for a particular class of diagrams, i.e. a visual language. During diagram construction, the editor parses the diagram, collecting geometric constraints which capture the semantics of the diagram. During diagram manipulation, objects in the diagram can be moved or resized while the constraint solver maintains the semantics of the diagram by preserving the geometric constraints between the diagram components.

Our interest in the intelligent diagram metaphor arises because it provides a much better basis for diagram beautification than does the traditional uninterpreted diagramming model. First, since the graphic editor is specialized for a particular visual language, it can provide visual language specific beautification rules. Second, parsing ensures that the semantics of the diagram are available to the beautification process. Finally, the underlying constraint solver provides the ideal technology for implementing beautification.

In this paper we describe a single extension to the Penguins system which supports diagram beautification. The Penguins system [2, 4] is intended to facilitate the development of diagram editors embodying the intelligent diagram metaphor by automatically generating a specialized editor from a grammatical specification of the visual language it is to support. Our extension supports beautification by allowing the grammar writer to specify "layout" constraints in the grammar productions. These constraints are not enforced during



(a) initial binary tree **(b) complete beautification** **(c) partial beautification**

Figure 1. Binary tree diagram

recognition or manipulation but are used when the diagrammer asks for the diagram (or some part of it) to be beautified. To ensure that there is never a conflict between the semantic constraints and the layout constraints, the layout constraints are treated as desired, rather than required. We provide a preliminary evaluation of this approach by describing three languages we have extended: Binary trees, mathematical equations and state transition diagrams.

To our knowledge there has been relatively little work on diagram beautification. One notable exception is that of Kurlander and Feiner [5] who allow the user to specify constraint-based search and replace rules. These can, for instance, match near right-angles and replace them by true right-angles. Our approach is both more and less powerful: Their rules do not allow global interaction between components so could not beautify trees but they do allow more general transformations such as replacing squares by circles which change the semantics. Furthermore the rules cannot easily take into account the syntax of the diagram. We also note the influence of Pavlidis and Van Wyk's illustration beautifier [6]. This uses several pre-defined rules, such as "make nearly coincident vertices coincident," to beautify arbitrary diagrams. Our system is considerably more powerful.

Other related work has focused on diagram layout rather than diagram beautification. (The difference between the two is one of degree and arises from their different purpose: Layout requires components of the diagram to be positioned from scratch while beautification starts with an initial layout and performs minor perturbations to improve it while still preserving the "feel" of the original layout). Both tree and graph layout have been extensively investigated (see [7] for an overview). The most closely related work has been layout in interactive settings in which case the new graph-layout should preserve the *user's mental model* [8]. The main difference to the present paper is that we consider general diagrams rather than just trees or graphs.

2. Motivating Example

We wish to develop tools for beautifying the appearance or "layout" of a diagram. In this section we illustrate by means of a running example the requirements on such a tool. Imagine that the rather unaesthetic binary tree in Figure 1(a) has been input by a user and is now ready to be beautified. Generally speaking we would like a diagram's layout to be clear, regular and uncluttered. Specific rules to achieve this in the case of binary trees are:

- Children should have the same vertical distance from their parent.
- A parent should be midway between children.
- The children of a node should not overlap.
- The tree should not be too wide.

Figure 1(b) shows the results of applying the specified beautification rules to the tree in Figure 1(a). Notice how the beautification rules preserve the original binary tree's semantics, that is, the parent and child relationship. We note that the beautification rules are necessarily binary tree specific: Indeed we believe that most beautification is visual language specific.

Often the user will have good reasons for where they place some components of the diagram and will not wish beautification to change this part of the diagram. Ideally, we would like the user to be able to select a sub-part of the diagram they wish to beautify. For example, if the user had selected nodes f, g, h, i and j in Figure 1(a) and then applied the beautification procedure, the result should be as shown in Figure 1(c). The selected sub-diagram now has all the properties that were desired of a "beautiful" binary tree, but because the user has not selected the entire diagram for beautification, the nodes that were not selected are left exactly where the user initially placed them.

Another point to note is that once the diagram has been beautified, the user should still be free to manipulate the diagram as they wish. Beautification should not restrict subsequent manipulation of the diagram.

3 Background

The main focus of the current paper is how we have extended the Penguins system so as to provide constraint-based tools for improving the layout of diagrams. To understand the Penguins system and in particular this extension, the reader needs a working knowledge of constraint multiset grammars and QOCA. We now describe these.

The Penguins system [2, 4] is a new system designed to facilitate the development of software that supports the intelligent diagram metaphor by automating most of the software development process, much like tools such as LEX and YACC automate the compiler construction process. The Penguins system takes a high level specification of a visual language and automatically generates a diagram editor which supports the creation and manipulation of diagrams in that particular visual language. The specification language is based on *constraint multiset grammars*. From the specification, the *parser generator* generates an incremental *diagram parser* or *spatial parser* which is incorporated into the standard *diagramming editor*. This diagramming editor provides standard graphic primitives, such as lines, circles, text and arrows. In order to provide support for free-hand drawing from a pen as an alternative input method, a *tokenizer* is also provided. A *constraint solver*, QOCA, is incorporated into the graphic editor to provide the constraint solving mechanisms necessary for geometric error correction and diagram manipulation. New terminal types can be added to provide language specific graphic primitives. The generated C++ code, together with the application specific routines is compiled to give the final intelligent diagram editor.

Constraint multiset grammars provide a high-level and declarative framework for the definition of visual languages. Terminal types in constraint multiset grammars refer to graphic primitives, such as *line* and *circle*, instead of the usual string tokens. Each *symbol type* has a set of one or more attributes, typically used to describe the properties of the symbol type. A *symbol* is an instance of a symbol type. The declarations for each symbol type must be given in the grammar. Those for a grammar for binary trees are:

```
declare symboltype bintree(
  left:bintree, right:bintree, root:node
) : starttype;
```

This specifies that the symbol type `bintree` is a non-terminal symbol and that it has three attributes: Its left branch, its right branch and its root node. Note that `node` is a terminal symbol which has three connectors, `up`, `down1` and `down2`, and attributes `height` and `width`, as shown in Figure 2. Each connector has two attributes, `x` and `y`, corresponding to its horizontal and vertical coordinate respectively.

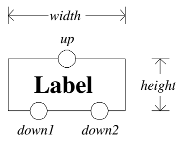


Figure 2. Attributes of a `node` symbol

The bulk of a constraint multiset grammar specification consists of productions, each describing how a non-terminal symbol can be recognized from other symbols. Productions have the form

$$P ::= P_1, \ldots, P_n \textit{ where } C$$

indicating that the non-terminal symbol P can be recognized from the multiset of symbols $P_1, \ldots, P_n$ whenever the attributes of all symbols satisfy the constraints C. The constraints enable information about spatial layout and relationships to be naturally encoded in the grammar. As an example consider binary trees.

Recognition of a binary tree is performed from the leaves upward, by first recognizing leaf nodes as `bintree` symbols and in turn, these `bintree` symbols are combined with their parent node to form other `bintree` symbols. This process is repeated until the root of the binary tree is reached.

The first production specifies a binary tree with two children. The parent node `P` connects to two children `C1` and `C2` with arrows `L1` and `L2` respectively. Notice that the constraints between the various components of the binary tree are rather loose, allowing flexibility in diagram creation. For example, a branch `C1` is considered to be the left child branch of `P` as long as it is placed on the lower left of `P`.

```
B:bintree() ::= P:node, L1:arrow,
      C1:bintree, L2:arrow, C2:bintree where (
    % Constraints on the left child.
    P.down1 == L1.start &&
    C1.root.up == L1.end &&
    P.down1.y <= C1.root.up.y &&
    P.down1.x >= C1.root.down2.x &&

    % Constraints on the right child.
    P.down2 == L2.start &&
    C2.root.up == L2.end &&
    P.down2.y <= C2.root.up.y &&
    P.down2.x <= C2.root.down1.x
  ) {
    % Initialize attributes.
    B.root = P;
    B.left = C1;
    B.right = C2;
}
```

The next production recognizes a leaf node or an isolated node. This is a node without an arrow starting from either of the connectors `down1` and `down2` of the node `N`. This condition is encoded using a negative constraint.

```
B:bintree() ::= N:node where (
    not exist A1:arrow where
      (A1.start == N.down1) &&
    not exist A2:arrow where
      (A2.start == N.down2)
  ) {
    B.root = N;
    B.left = 0;
    B.right = 0;
}
```

Negative constraints are commonly used to remove ambiguity in parsing. For example, if the negative constraints are not specified, it is impossible to know whether to use this production or the previous production.

One feature of constraint multiset grammars which is not utilized in the binary tree grammar is *existential quantification*. Expanding the constraint multiset grammar formalism in this way essentially makes it context-sensitive.

Constraint solving is at the heart of the Penguins system and crucial in diagram beautification. We use the C++ constraint solving toolkit QOCA which has been expressly developed for interactive graphical applications such as diagram editing [9]. QOCA provides three main components: Constrained variables, called `CFloats`, linear arithmetic constraints, called `LinConstraints`, and constraint solvers, called `Solvers`.

To the application programmer the `CFloat` appears to have a single floating point value, which they set using the method `SuggestValue` and then read after calling the constraint solver using `GetValue`. Internally, however, the desired and actual values are kept separately. It also has a *stay weight* and an *edit weight* which, respectively, indicate the importance of leaving the variable unchanged if it is not being edited and the importance of changing it to the desired value when the variable is being edited. Both weights are non-negative floats and are set when the `CFloat` is created and cannot subsequently be changed.

For the binary tree example, we use `CassSolver` which supports linear arithmetic equalities and inequalities. Such arithmetic constraints are the minimum necessary to support geometric relationships such as "to the left of" and "these points are coincident." The solver is based on the Cassowary constraint algorithm described in [10]. For the other two examples, we use `LinEqSolver`, which is faster but supports only linear equalities.

Each solver in QOCA provides the following methods for constraint manipulation. Constraints can be added to the solver one at a time using `AddConstraint`. With each addition the solver checks that the new constraint is compatible with the current constraints. This method is used to add constraints which enforce the geometric relationships found in parsing. `RemoveConstraint` allows the removal of a constraint which is currently in the solver. This is used when an object is deleted by the user or when invalidation of a negative constraint causes parsing to be undone.

Neither of these methods computes new values for the `CFloats`. The method `Solve` does this. It finds values which satisfy the current constraints but which are as close as possible to the desired values. This is used for error correction.

In addition, the solver provides methods for direct manipulation, however these are not of concern here, and more details can be found in [9].

4. Semantics based beautification

In this section we explain how the Penguins system has been extended to provide visual language dependent diagram beautification of the kind illustrated in Section 2.

The first point to note is that the Penguins system already performs some automatic beautification of the diagram during its construction. During parsing, equality constraints in a production are regarded as being satisfied by the symbols being matched if they are within a certain error tolerance. This is necessary since it is almost impossible for the user to precisely place objects in a diagram, especially when drawing with a stylus. Once the parser has determined that a particular production is applicable, the associated constraints (including the equality constraints) are added to the constraint solver and the constraint solving method is called to find new values for the variables which are as close as possible to the old values (i.e. the values they now have in the diagram) but which satisfy all of the constraints in the solver. The diagram is then redrawn to reflect the new symbol placement, providing feedback to the user about what has been recognized.

However, such automatic beautification does not significantly change the diagram's appearance, rather it is a form of *error correction* which forces the diagram to conform to the syntax of the underlying visual language. It cannot be used to provide the kind of beautification discussed in Section 2. Even if we modified the underlying visual language from binary trees to be that of "beautiful binary trees" this would not provide the user with much help since the editor would now only recognize trees that were already beautifully laid out.

Nonetheless, we can use essentially the same idea behind error correction to provide true diagram beautification. Basically we can add extra "beautification" constraints to the solver, call the solver to find new values for the variables which are as close as possible to the old values (i.e. the values they now have in the diagram) but which satisfy all of the constraints in the solver. We then redraw the diagram and remove the beautification constraints from the solver. For this idea to work in practice, however, two issues need to be addressed.

First, we wish the user to be able to leave some of the diagram unchanged. Because of the presence of semantic constraints which may link variable values in unexpected ways, the only sure way to guarantee that the beautification process does not change a variable, say x, from its current value, say c, is to temporarily add an *anchor* constraint $x = c$ to the constraint solver during beautification.

The second issue arises because we have no way of guaranteeing that the beautification constraints are compatible with the constraints already in the constraint solver. This is exacerbated by allowing anchor constraints. Thus, during beautification we might end up with a collection of constraints which is unsatisfiable. We can overcome this problem by treating the beautification constraints as *desired constraints*. Such constraints are not required to be satisfied, but should be satisfied when possible. Constraint hierarchies [11] formalize such preferences.

During beautification we would like to be able to combine the conflicting desired constraints and find the solution which globally minimizes the conflict, taking into account the importance or *weight* of the beautification constraints. In order to do this we associate with each desired constraint c an *error function* ϵ over the variables in c which returns the error associated with a particular assignment to the variables. The error is required to be a non-negative real number and to be 0 if the constraint is satisfied. For instance, the error associated with an equation $s = t$ might be $|s-t|$, and for an inequality $s \leq t$ which is more important, it might be $1000|s-t|$ when $s > t$ and 0 otherwise.

The use of desired constraints has another advantage. By appropriate scaling of the error functions it allows the user to determine the relative importance of changing the variable values as little as possible, so preserving the current diagram, versus the importance of satisfying the beautification constraints and so getting a better layout.

Let us make this whole discussion a little more formal. At any point in time the editor has a current *configuration* which consists of:

- A parse tree, P.[1]
- A set of variables $\{v_1, ..., v_n\}$ which correspond to the spatial attributes of the terminal symbols.
- An assignment $\{v_1 \mapsto c_1, ..., v_n \mapsto c_n\}$ to those variables which corresponds to the current diagram layout.
- A conjunction of constraints C over these variables which were added in parsing and which capture the semantics of the diagram.

Now imagine that the user has selected the graphic objects $t_1, ..., t_m$ in the current diagram and asked their layout to be beautified with positive strength w. Then the beautification process uses the constraint solver to find values $c'_1, ..., c'_n$ for $v_1, .., v_n$ which minimize

$$w(\sum_{j=1}^{p} \epsilon_j) + (1/w)(\sum_{i=1}^{n} |v_i - c_i|)$$

subject to the constraints

$$C \wedge \left(\bigwedge \left\{ v_i = c_i \;\middle|\; \begin{array}{l} 1 \leq i \leq n \text{ and } v_i \text{ does not} \\ \text{correspond to an attribute of} \\ \text{some } t_k \end{array} \right\} \right)$$

where $\epsilon_1, ..., \epsilon_p$ are the error functions of the beautification constraints.

After beautification the current configuration remains the same except that the assignment becomes $\{v_1 \mapsto c'_1, ..., v_n \mapsto c'_n\}$ and the diagram must be redrawn to reflect this "beautification."

Note that the constraints in the above optimization problem are always solvable since $\{v_1 \mapsto c_1, ..., v_n \mapsto c_n\}$ is a solution. (By assumption the current solution always solves the current constraints.) Furthermore, the optimization problem is always bounded since it contains the term $(1/w)(\sum_{i=1}^{n} |v_i - c_i|)$. Thus, the above optimization problem always has a solution.

We also note that only for those variables v_j corresponding to attributes in the graphic objects $t_1, ..., t_m$ can $c_j \neq c'_j$. Thus the unselected portion of the diagram will remain unchanged by beautification.

The final issue is how do we specify the beautification constraints and their corresponding error functions. As we indicated in Section 2, beautification should take into account the kind of diagram we are dealing with, that is, its syntax. In particular, we need access to the diagram syntax in order to know how to

[1]Actually parse trees $P_1, P_2, ...$ but for simplicity we assume there is a single parse tree.

beautify a particular element. For instance, text labelling an arrow in a state transition diagram might have very different beautification constraints applied to it than does text labelling a state in the same diagram. For this reason, it is reasonable to associate the beautification constraints with productions in the grammar, thus allowing the constraints to be applied to the appropriate syntactic elements in the parse tree.

As an example we consider how beautification constraints can be added to the productions in the grammar for binary trees. For the purpose of beautification, we need to introduce two "layout attributes" to the `bintree` symbol type: `leftmargin` and `rightmargin` which are the left and right boundaries of a branch respectively. The amended declaration of the symbol type `bintree` is:

```
declare symboltype bintree(
  left:bintree, right:bintree, root:node,
  layout leftmargin:integer,
  layout rightmargin:integer
) : starttype;
```

We must now place beautification constraints in each of the binary tree grammar's production. First, in the context of a binary tree with two children, beautification is achieved by ensuring that the spacing between the nodes is visually pleasing. The first step is to make sure that children do not overlap. To do this, we make the right margin of the left child and the left margin of the right child the same. Further beautification can be achieved by making the horizontal distance from the two children to the parent equal. We also place adequate vertical distance between the two children and the parent.

```
B:bintree() ::= P:node, L1:arrow,
      C1:bintree, L2:arrow, C2:bintree where (
    ...
} layout {
  % Set the left and right margins.
  B.leftmargin == C1.leftmargin - 5;
  B.rightmargin == C2.rightmargin + 5;

  % Ensure no overlapping.
  C1.rightmargin == C2.leftmargin;

  % Equal horizontal distance from children to parent.
  P.down1.x - C1.root.down2.x
      == C2.root.down1.x - P.down2.x;

  % Vertical separation equal the height of parent.
  C1.root.topleft.y - P.topleft.y
      == 2 * val(P.root.height);
  C2.root.topleft.y - P.topleft.y
      == 2 * val(P.root.height);
}
```

The keyword `val(`*a*`)` in a beautification constraint obtains the value of the attribute a instead of adding a as part of the constraint. This allows a to be used as a constant in a constraint.

Second, in the context of a leaf node or an isolated binary tree, only constraints for left and right margins need to be specified. The following is the amended production.

```
B:bintree() ::= N:node where (
    ...
} layout {
  % Set the left and right margins.
  B.leftmargin == N.topleft.x - 5;
  B.rightmargin == N.topleft.x + val(N.width) + 5;
}
```

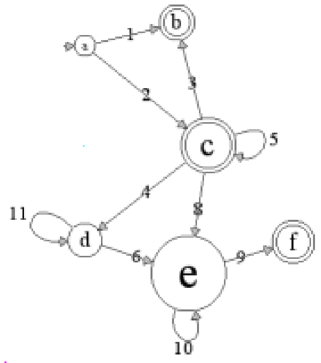
The remaining two productions for a binary tree are modified in a similar fashion.

Notice how in the above example we used attributes to pass information about layout within the parse tree. This allows beautification to behave globally rather than just locally.

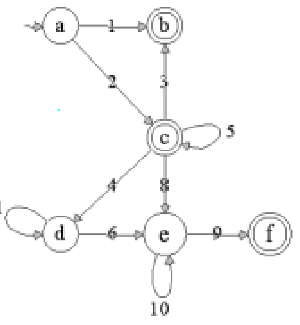
The actual mechanism for specifying beautification constraints is somewhat more powerful than indicated in the above example. We can also provide *conditional* beautification constraints. These are only applied if the elements in the current diagram satisfy the condition. An example of such a rule is used in the following production rule for recognizing a transition in a state transition diagram.

```
T:transition() ::= A:arc where (
    exist S1:state, S2:state where (
      StartOnCircle(A, S1.mid, S1.radius) &&
      EndOnCircle(A, S2.mid, S2.radius)
    )
  ) {
    T.start = S1.label;
    T.label = A.label;
    T.end = S2.label;
} layout {
  if (fabs(A.start.y - A.end.y) < 30)
    A.start.y == A.end.y; % Make horizontal.

  if (fabs(A.start.x - A.end.x) < 30)
    A.start.x == A.end.x; % Make vertical.

  S1.radius == S2.radius; % Equal radius.
}
```

If the arc `A` is close to horizontal then a beautification constraint stating that the y coordinates of the start and end points should be made equal is added. An analogous beautification constraint is put in place if the arc `A` is close to vertical. Another beautification constraint specifies that the radius of both states `S1` and `S2` should be made equal ensuring that all states in a connected state transition diagram will have the same radius.



(a) initial state transition diagram



(b) beautified state transition diagram

$$x^2+2x-8=(x-2)(x+4)$$

(c) initial mathematical equation

$$x^2+2x-8=(x-2)(x+4)$$

(d) beautified mathematical equation

Figure 3. Other beautification examples

There are several issues arising in the implementation. The first issue is how to generate the beautification constraints when asked to perform beautification on the diagram. As we have indicated, during diagram construction, a parse tree is built which captures the semantics of the diagram. All leaf nodes in the parse tree correspond to terminals and all internal nodes correspond to non-terminals. Each non-terminal has an associated creation production P and the associated beautification constraints specified in P. When performing a diagram beautification, we simply add the beautification constraints to the constraint solver which are associated to each of the non-terminals in the parse tree.

When performing beautification on a selected section of the diagram, only related beautification constraints need to be added to ensure efficiency in diagram beautification. This can be done by simply adding only the beautification constraints associated to the ancestor non-terminals of the selected terminals. In order to ensure that the position of the unselected terminals remain the same, anchor constraints are generated for each geometric attribute of the unselected terminals.

Another issue is that the underlying constraint solver, QOCA, does not directly support desired constraints, only desired values for variables. We overcome this by introducing an auxiliary variable for the error and setting its desired value to be 0. For instance, the desired constraint $s = t$ is actually handled by adding a new `CFloat` e to the solver with desired value 0 and then adding the (required) constraint $s - t = e$ to the solver. The stay weight for e is chosen to reflect the desired constraint's strength and the strength of beautification. The case for an inequality $s \leq t$ is similar. We simply add a new `CFloat` e to the solver with desired value 0 and add the required constraint $s \leq t + e$.

5. Evaluation

We have implemented the extension to the Penguins system and tested it with three representative visual languages: Binary trees, state transition diagrams and mathematical equations. In particular, the diagrams shown in the motivating example of Section 2 are those actually resulting from using the system for binary tree layout when using the beautification constraints described in the example grammar.

Beautification of state transition diagrams is quite simple. The beautification constraints attempt to make the size of all the connected states uniform, and also to make near vertical or horizontal transition arrows vertical or horizontal. Encoding the latter beautification rule requires the use of the conditional layout construct. The actual production is listed in Section 4. An example beautification of a state transition diagram is shown in Figure 3(a) and Figure 3(b).

The third example is beautification for mathematical equations. Consider the simple equation in Figure 3(c). We desire the following attributes in an equation for it to be considered well laid out. First, the positioning of the symbols should clearly depict the relationship in the equation, i.e. the symbols need to be spaced to give an indication of relationships to each other and to the operators. For example, two symbols that are to be multiplied cannot be too far apart or the meaning of this relationship is lost. It is also important that exponents are not confused with multiplication and vice versa. Secondly, the symbols in the equation should have equal sizes, except in the case of the exponent which should have a smaller size to reflect its relationship to the base symbol more clearly. Also, the operators in the equation should also have equal sizes that are proportional to the size of the symbols. Lastly, the left and right brackets in the expression should be equally sized and where possible, uniform throughout the entire equation.

The grammar specifies some of these attributes as recognition constraints, such as the relative positioning of symbols. The beautification constraints specify the desired relative sizing of the elements. The result of beautification shown in Figure 3(d) shows a definite improvement but it is still not ideal, mainly due to way that the different fonts used place their symbols with respect to the bounding boxes and baselines of the font.

6. Conclusion

We have argued that the intelligent diagram metaphor naturally supports diagram beautification which is visual language dependent and which preserves the semantic relationships in the diagram. This is achieved by modelling the semantic relationships using required geometric constraints and adding beautification constraints as desired constraints. As a proof of concept we have extended the Penguins system so that the programmer can add beautification constraints to the constraint multiset grammar. The Penguins system will then generate an editor for the visual language specified by the grammar which provides a beautification function with which a user of the editor can beautify selected portions of their diagram. We have tested this using three visual languages: Binary trees, state transition diagrams and mathematical equations, and the results are very encouraging.

The current performance of our system is adequate for small or medium sized diagrams. Currently it takes about four seconds on a Dual Pentium-II 450 MMX running Microsoft Windows NT 4.0, to beautify the entire binary tree given in Figure 1(a), while beautification of the state transition diagram in Figure 3(a) and the mathematical equation in Figure 3(c) take less than one second. As an indication of the number of constraints involved, we note that the binary tree diagram has 226 required and 50 layout constraints. Most of beautification time is spent inside the constraint solver QOCA. We believe it is possible to speed up execution by extending QOCA to provide a method for adding a number of constraints as a batch. Currently constraints must be added one at a time and QOCA checks for satisfiability after each addition. Adding all of the constraints for beautification and only then performing the test for satisfiability should be considerably faster.

The current system, although quite flexible, does not allow arbitrary beautification. We would like to add transformation rules to the Penguin system. These would extend the current approach by allowing the beautification process to change the syntactic nature of the item being beautified.

Another expressiveness limitation arises from the underlying constraint solver and its restriction to linear constraints. We would also like to allow user-defined constraints and function in the layout part of the grammar as this allows the grammar writer to encode arbitrary layout conditions, especially by judicious use of attributes. One point to note is that such user-defined constraints must still behave as desired constraints—they can only suggest values for variables since they must not conflict with the semantic constraints.

References

[1] L. Weitzman and K. Wittenburg, “Relational grammars for interactive design”, in *IEEE Symposium on Visual Languages*, 1993, pp. 4–11.

[2] S.S. Chok and K. Marriott, “Automatic construction of user interfaces from constraint multiset grammars”, in *IEEE Symposium on Visual Languages*. 1995, pp. 242–250, IEEE Computer Society Press.

[3] M.D. Gross and E.Y.-L. Do, “Ambiguous intentions: a paper-like interface for creative design”, in *9th ACM Symposium on User Interface Software and Technol ogy (UIST)*. 1996, pp. 183–192, ACM Press.

[4] S.S. Chok and K. Marriott, “Automatic construction of intelligent diagram editors”, in *Proceedings of the 11th ACM Symposium on User Interface Software and Technology*. 1998, pp. 185–194, ACM Press.

[5] D. Kurlander and S. Feiner, “Interactive constraint-based search and replace”, in *ACM Conference Human Factors in Computing (CHI)*. 1992, pp. 609–618, ACM Press.

[6] T. Pavlidis and C. Van Wyk, “An automatic beautifier for drawings and illustrations”, in *Proceedings of SIGGRAPH’85*, 1985, pp. 225–234.

[7] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, New Jersey, 1999.

[8] K. Misue, P. Eades, W. Lai, and K. Sugiyama, “Layout adjustment and the mental map”, *Journal of Visual Languages and Computing*, vol. 6, pp. 183–210, 1995.

[9] K. Marriott, S.S. Chok, and A. Finlay, “A tableau based constraint solving toolkit for interactive graphical applications”, in *International Conference on Principles and Practice of Constraint Programming (CP98)*, 1998, pp. 340–354.

[10] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao, “Solving linear arithmetic constraints for user interface applications”, in *Proceedings of the 10th ACM Symposium on User Interface Software and Technology*. 1997, pp. 87–96, ACM Press.

[11] A. Borning, B. Freeman-Benson, and M. Wilson, “Constraint hierarchies”, *Lisp and Symbolic Computation*, vol. 5, no. 3, pp. 223–270, Sept. 1992.