Attribute Grammars in Constraint-based Graphics Systems

Lee Alton Barford*

Bradley T. Vander Zanden†

87-838

June 1987

Department of Computer Science

Cornell University

Ithaca, New York 14853-7501

*The work of this author was supported by a Cornell Mathematical Sciences Institute Fellowship, by the National Science Foundation under grants DCR-8502568 and DMC-8312096, and by the Office of Naval research under grant N00014-86-K-0281.

†The work of this author was supported by the National Science Foundation under grants 86-02663 and 85-03611.

Attribute Grammars in Constraint-based Graphics Systems

Lee Alton Barford[*]

Bradley T. Vander Zanden[†]

June 1, 1987

Abstract

A constraint-based graphics system provides a flexible, intuitive framework for describing relationships among graphical objects in applications such as document preparation, font design, and solid modelling. This paper describes two constraint-based graphics systems, micro-COSM and the IDEAL Synthesizer, and their implementation in terms of attribute grammars. The implementation of these two systems is noteworthy since they represent the first interactive constraint-based graphics systems that are implemented using attribute grammars. Our experiences with attribute grammars suggest that they provide a powerful framework for representing constraints and extracting important semantic information such as the equations to be solved by the constraint solver. We discuss the advantages of using attribute grammars in constraint-based graphics and from our experiences make several observations about the way attribute grammars should be used.

1 Introduction

Constraint-based graphic systems provide an extremely flexible and intuitive user interface for such diverse applications as document preparation [1], font design [2], and solid modelling [3]. In

[*]The work of this author was supported by a Cornell Mathematical Sciences Institute Fellowship, by the National Science Foundation under grants DCR-8502568 and DMC-8312096, and by the Office of Naval Research under grant N00014-86-K-0281.

[†]The work of this author was supported by the National Science Foundation under grants 86-02663 and 85-03611.

a constraint language, a geometric primitive, such as a point or a sphere, is represented as a set of real variables that specify the primitive's size and location. Constraint equations can be used to relate these variables in any desirable manner, thus providing flexibility in expressing relationships between primitives.

Several fundamental issues must be addressed when implementing constraint-based graphics systems. Among these are:

1. User interface issues.
2. Constraint language design.
3. Internal representation of the constraints.
4. Error checking and validation of constraints.
5. Extraction of important semantic information from the constraints' representation.
6. Constraint satisfaction.

In this paper we describe how these issues were addressed in two constraint-based graphics editors implemented by the authors. One is the constraint-based solid model editor, microCOSM [3]. The other is the IDEAL Synthesizer, an interactive dialect of Van Wyk's IDEAL language [1] using the Cornell Synthesizer Generator [4].

The user interface issues and the design of the constraint language are an integral part of the two systems and are fully discussed in the paper. However, the most noteworthy aspect of the two systems is that they use attribute grammars to implement items 3-5 above, and in the case of the IDEAL Synthesizer, item 6 as well. Until recently, it was not feasible to implement interactive applications using attribute grammars since offline algorithms were the only known method for evaluating them. However, the introduction of incremental evaluation algorithms has made it feasible to implement interactive language-based editors via attribute grammars [5,4]. The IDEAL Synthesizer owes its existence to this algorithm. In contrast, the microCOSM editor uses an offline algorithm but organizes the application so that the algorithm is never given a large input.

We chose attribute grammars to implement our systems since they have three important advantages over alternative, procedural approaches (e.g., semantic-action routines [6]) that might have implemented our systems just as efficiently [5]:

1. Attribute grammars are applicative: The semantic information associated with each production is encoded as a set of equations (the variables that comprise these equations are termed attributes). Implicit in the formalism is the notion that equations propagate semantic information to the parts of the application that need it. This propagation is achieved simply by invoking an equation solver. In contrast, a procedural approach requires that each production specify the propagation mechanism explicitly via procedures. In practice, each of these procedures specify an elaborate mechanism that walks the entire tree derived from the application.

2. Attribute grammars encourage a modular specification: In attribute grammars, the semantic equations associated with a production are encoded solely in terms of the attributes that are declared in that production. In contrast, a procedural approach requires that the structure of part or all of the tree be known in order to compute the semantic information associated with a given production.

3. Attribute grammars have an automatic "undo" facility: Since the semantic information associated with the application is encoded in equations, an action can be undone simply by restoring the original set of equations and invoking the equation solver. In contrast, a procedural approach requires that an explicit rollback procedure be associated with each production since the original action may have induced side-effects.

The remainder of the paper is organized as follows. Section 2 introduces terms and terminology associated with constraint-based graphics editors and with attribute grammars. Section 3 describes the language design of the microCOSM language, the graphics- and language-based editor for microCOSM, and the implementation of microCOSM in terms of attribute grammars. Section 4 does the same for the IDEAL Synthesizer. The lessons we learned in implementing our systems and later, by comparing them, are presented in section 5. The paper is summarized in Section 6.

2 Definitions and terminology

2.1 Constraint Languages and Editors

Both microCOSM and IDEAL are called constraint languages. As indicated in the introduction, a constraint language represents geometric primitives, such as a point or a sphere, as a sequence of real variables that specify the primitives' size and location. Functional relationships between primitives can be expressed as **constraint equations** (or usually just **constraints**). For example, the relationship that two points (x, y, z) and (u, v, w) are distance 17 apart is given by the equation

$$(x-u)^2 + (y-v)^2 + (z-w)^2 = 17^2.$$

A **constraint solver** finds values for the variables so that the constraints hold and then the editor displays or prints the primitives in the locations given by the variable values. The constraint solver used with a particular constraint language depends on the nature of the constraints permitted by the language. If the language allows only linear equations, Gaussian elimination can be used to solve the constraints. If multivariate polynomials are allowed, more sophisticated numerical methods must be used.

Constraint languages are amenable to graphical editing — editing a program by editing a display of the primitives. For example, the user might be allowed to modify variable values by selecting and stretching primitives. In other systems, the user can change variable values by changing the text of the program. In either case, after the values are changed, some of the constraint equations may no longer be satisfied. The editor uses the constraint solver to find new values for the variables that once again satisfy the constraints. The resulting modified object is then redisplayed.

Using constraints to represent relationships between primitives in a two-dimensional drawing is one of the oldest ideas in computer graphics. Sutherland's Sketchpad [7] allowed the user to draw finite-extent primitives such as line segments and circular arcs. The size and location of the primitives were determined by floating-point variables. The user could establish linear constraints on these variables. When the user selected and moved a primitive, a relaxation method was used to find an approximate solution to the resulting system of linear equations.

ThingLab [8] was the first constraint-based editor that was really a language-based editor for an underlying textual language. The language provided a class hierarchy similar to Smalltalk's [9]. Primitive classes included real numbers and finite sets. Constraints were solved by repeatedly invoking procedures given by the user or by linear relaxation.

2.2 Attribute Grammars

An **abstract syntax tree** [10, p. 49] is essentially a parse tree. A parse tree has a vertex for every token in the source file and every use of a production in the parse. An abstract syntax tree usually has instead one vertex for each identifier, numeric constant, and production. The word "abstract" is used because syntactic details such as keywords and punctuation do not appear in the tree. An abstract syntax tree can be thought of as a parse tree for a grammar that has had these details removed. Such a grammar is called an **abstract grammar**. The abstract syntax tree for the program fragment in Fig. 3 is shown in Fig. 1.

Attributes [10, Ch. 5] are values that are computed and stored at each vertex of an abstract syntax tree. The value of an attribute at a vertex is a function of the values of the other attributes at the vertex and at all the vertex's neighbors. The function used to compute an attribute at a vertex depends on the production represented by that vertex. A description of the productions of an abstract grammar along with the functions to compute the attributes for each production of the grammar is called an **attribute grammar**.

Attributed abstract syntax trees are used to represent programs in language-based editors. A single modification of a program represented by an abstract syntax tree corresponds to a replace-

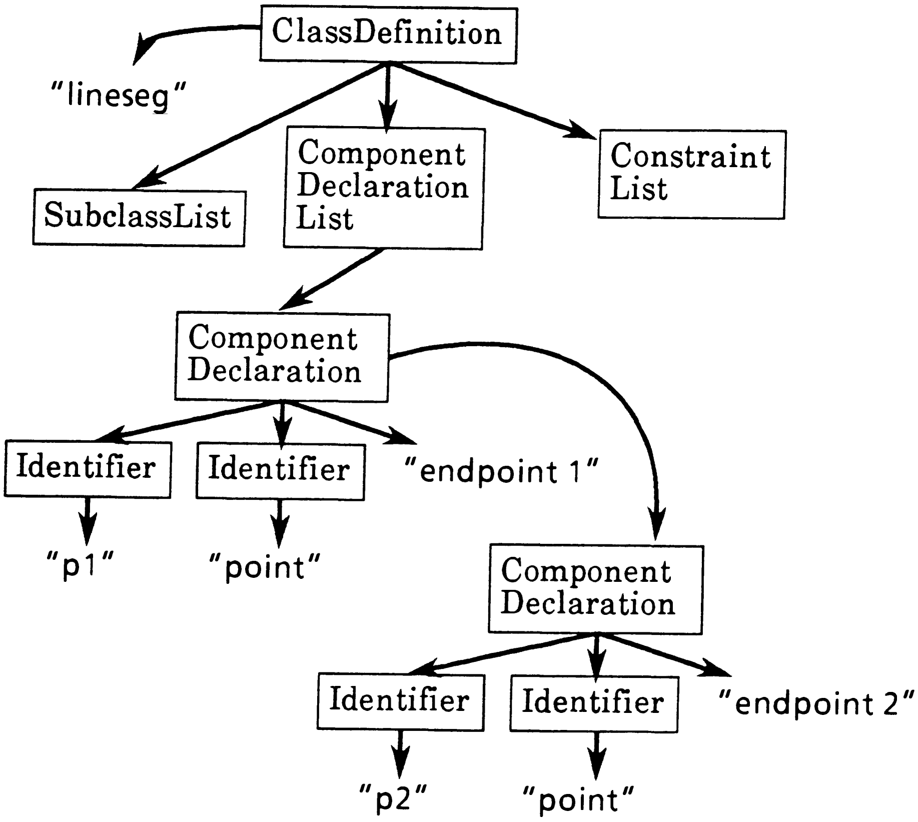

Figure 1: Abstract syntax tree for class **`lineseg`**

ment of some subtree of the abstract syntax tree by another tree.

After the abstract syntax tree is modified, some of the attributes of the abstract syntax tree will no longer have correct values and will have to be recomputed. Often, however, the change will only affect the attribute values in an area of the abstract syntax tree near the site of the change. The problem of how to take advantage of this locality to minimize the number of attributes that are recomputed after an editing operation is an extremely active area of research. [11] is the seminal work in this field.

3 microCOSM

A solid model is a representation of the space occupied by a rigid object. A generic solid (or "generic") expresses the solid models for a class of similar objects, or equivalently, a non-rigid object. Automated manufacturing systems require solid models of the objects with which they work. For example, solid models are needed to discover collision-free paths for robot arms.

microCOSM is a constraint language specifically designed for expressing definitions of generic solids. These definitions can be edited using either an ordinary text editor or the microCOSM editor. The microCOSM editor is a graphical, language based editor. A microCOSM definition is edited by performing graphical operations (menu selection, dragging,...) on a picture representing the definition. These editing operations are ultimately reflected by changes in the textual form of the definition. The language will be introduced first.

3.1 The microCOSM language

In microCOSM, generic solids (and generic non-solids, such as lines and planes) are called **classes**. A class, then, is a specification for a set of solid models of similar objects. Each of these solids is called a **member** of the class. A class is defined by

- the parts (called **components**) of members of the class (which are themselves members of other classes), and

```
CONSTRAINT PtId "Identify two points" ON
     p1 :  point "a point to make same as another";
     p2 :  point "the point to make the first one the same as";
CONSTRAINED BY
     p1.x = p2.x;
     p1.y = p2.y;
END
```

Figure 2: Constraint definition for point identification

```
CLASS lineseg "2-d line segment" IS
     p1 :  point = ( x= 0.1, y= 0.1 ) "an endpoint";
     p2 :  point = ( x= 0.2, y= 0.4 ) "another endpoint";
END
```

Figure 3: Class definition for line segments

- constraints on the components.

The only primitive class in microCOSM is the class `real` of floating point numbers. The primitive constraints provided by microCOSM are polynomial equations in `real` variables. Constraints can also be defined as conjunctions of other constraints. Any class definition is thus equivalent to a system of polynomial equations in real variables.

Fig. 2 shows the definition of the constraint that two points occupy the same location. The first line says that the name of the constraint is `PtId`. Next are the declarations of formal arguments. `PtId` takes two points as arguments, `p1` and `p2`. The equation that follows constrains the x coordinates of the two points to be the same. The second equation does the same for their y coordinates.

Some simple class definitions are shown in Figs. 3-4. The definition in Fig. 3 states that line segments are determined by their endpoints, p1 and p2. The expressions following the component

```
CLASS quad "quadrilaterals" IS
     s1 :  lineseg = ( p1=(x=0.1, y=0.1), p2=(x=0.1, y=0.3))
                     "side #1";
     s2 :  lineseg = ( p1=(x=0.1, y=0.3), p2=(x=0.3, y=0.3))
                     "side #2";
     s3 :  lineseg = ( p1=(x=0.3, y=0.3), p2=(x=0.3, y=0.2))
                     "side #3";
     s4 :  lineseg = ( p1=(x=0.3, y=0.2), p2=(x=0.1, y=0.1))
                     "side #4";
CONSTRAINED BY
     PtId(s1.p2, s2.p1);
     PtId(s2.p2, s3.p1);
     PtId(s3.p2, s4.p1);
     PtId(s4.p2, s1.p1);
END
```

Figure 4: Class definition for quadrilateral

declarations give initial values for the components. They are called **initializers**. One endpoint will be at $(0.1, 0.1)$. The other endpoint will be at $(0.2, 0.4)$. When a class declaration is read into the microCOSM editor, the initial values give the size and position of an example object from the class.

In Fig. 4, **quad** is the definition of the class of quadrilaterals. The components of a quadrilateral are its four sides. `PtId` is used to force the sides to be linked into a cycle. The initializers given in **quad** for each side override the initial values for the endpoint coordinates given in `lineseg`.

The microCOSM language also has features necessary for specifying solid models. However, these features are not important for the purposes of this paper and will not be described here. (For more details, see [3].)

3.2 The microCOSM editor

The microCOSM editor allows microCOSM class definitions to be edited graphically, one at a time. The definition being edited is called the **primary definition**. A picture of the geometric object given by the primary definition appears in a graphics window. Each graphical editing operation on the picture corresponds to a change in the textual form of the definition. The graphical editing operations of the microCOSM editor will be presented by describing how the definition of **quad** in Fig. 4 is built using the microCOSM editor.

To create a new class, the user selects the function "New class" from a menu. The editor prompts for the name of the new class, and the user enters "quad." Class **quad** is initially the class with no components or constraints. That is, the text form of the initial definition is

```
CLASS quad "" IS
END
```

To create the rectangular body of the plate, the user selects the "Add Component" from the menu and types the name of the class of the component to add: "lineseg." (It would, of course, be much more convenient to have the user select the class from a menu of available classes. This capability will be added in the future.) A new component of class `lineseg` is added to the definition of **quad**, and the segment is displayed on the screen. The size and location of the segment can be changed by selecting either of its endpoints with the mouse and moving it. To add the other line segments to the class **quad**, the above "Add Component" operation is repeated three times.

Next, the constraints that connect together the endpoints of the segments are added. The user selects "Add Constraint" from the menu and types the name of the necessary constraint: "PtId." The editor prompts the user to "Pick a point to be made the same as another." The user selects the point with the mouse. Next, the editor prompts "Pick the point to make the first one the same as," and the user selects the other point. (The prompts for constraint arguments are obtained from the comments in the formal parameter declarations in the constraint definitions.) The editor adds the new constraint to the class definition of **quad** and moves the two points so that they coincide. The other three PtId constraints are added similarly.

3.3 Implementation

We will describe the implementation by first discussing the crucial data structures and then showing how those data structures are changed by the various editing operations.

3.3.1 Data Structures

At any time during an editing session, the picture in the editing window is determined by the microCOSM definition of the primary class, along with the current values of the variables that give the size and location of all the primary class's components. Editing operations are of two kinds. **Structural** editing operations, such as adding and deleting components and constraints, result in changes to the definition of the primary class. **Repositioning** operations, such as selecting and moving a primitive, result in changes to the current values of the variables.

The editor stores the microCOSM definitions of the primary class and of classes and constraints contained by the primary class as individual abstract syntax trees. The abstract syntax tree that gives the definition of the primary class is called the **primary abstract syntax tree** (or **PAST**). Together, the abstract syntax trees stored in the editor at any one time are called the **abstract syntax forest**. The current values of the variables are stored in the **current value array**.

3.3.2 Editing operations

A block diagram of the microCOSM editor is shown in Fig. 5. The clue to understanding the implementation of the microCOSM editor is the division of labor between the attribute evaluator and the constraint solver. The attribute evaluator:

- checks that the primary definition as encoded by the PAST contains no errors, such as undeclared identifiers or type errors,
- extracts information necessary for display (the primitive list) and solid modeling from the PAST,

- finds the number of variables required and extracts constraint equations to form the system of equations that the current values must satisfy.

The attribute evaluator does not affect the current value array. The constraint solver guarantees that the variables in the current value array satisfy the system of equations extracted from the PAST.

Class and constraint definitions are stored on mass storage as textual microCOSM programs. When the user requests that a new primary class be loaded, its textual program is parsed and the new PAST is constructed. The attribute evaluator then computes the values of the attributes in the PAST.

A structural editing operation corresponds to the addition or deletion of a subtree of the PAST. If a new component `x:real` were added to `lineseg`, the abstract syntax tree in Fig. 6 would be added to the abstract syntax tree in Fig. 1. The root of the new subtree would be made a member of the list of ComponentDeclaration vertices. After the PAST is modified, some of the attributes of the PAST will no longer have correct values and will have to be recomputed. Since the PAST tends to be relatively small, attribute reevaluation is done by brute force: all the attributes of the PAST are recomputed after every structural editing operation.

If the editing operation added new constraints, the constraint solver is called to make sure that the current values satisfy the constraints. The constraint solver uses a numerical method based on [12] to find a solution to the system of equations that is a closest solution to the values in the current value array. That solution is then placed into the current value array. Intuitively, choosing a closest solution guarantees that solving the constraints does not change the picture more than it has to. Finally, the graphical primitives are redisplayed in their new positions.

Repositioning operations can be handled somewhat more simply. When the user selects and moves part of the picture, the editor finds what variables correspond to the graphics primitives in the part of the picture being moved and changes the values of the variables to make the primitives appear in their new locations. If the current values no longer satisfy the constraints,

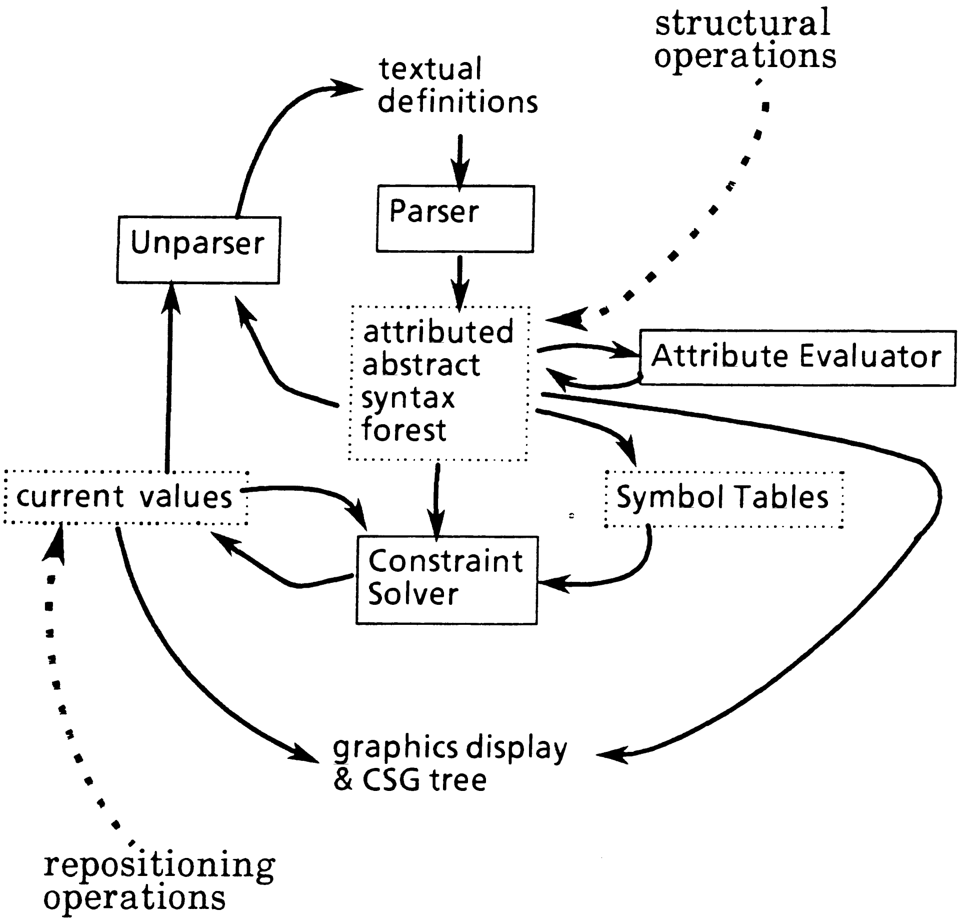

Figure 5: Block diagram of microCOSM editor

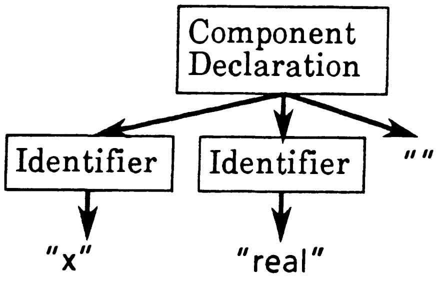

Figure 6: abstract syntax subtree for new component `x : real`

the constraint solver changes the current values so that the constraints once again hold. If the user has not finished modifying the picture, the above process is repeated. If constraint solving can be performed fast enough, this gives the user the illusion that the picture is changing as he moves the mouse. This is desirable since the user can explore the structure of the constraints on the primary object by observing how it responds to moving parts of the picture.

The current implementation of the microCOSM editor runs on a SUN Microsystems SUN-2 workstation, under the Berkeley 4.2 UNIX [1] operating system. The constraint solver contains a few FORTRAN routines from some standard numerical routine libraries.

4 The IDEAL Synthesizer

The IDEAL Synthesizer is an interactive language-based text editor that computes and displays graphical objects based on a textual specification input by the user. Users can interactively modify the graphical display by pointing with the mouse at the appropriate spot in the textual specification and modifying it. The graphical display is updated by using an incremental constraint solving algorithm that requires time proportional to the number of objects affected by the modification. In contrast, the original version of IDEAL uses an offline algorithm that recomputes the entire

[1] UNIX is a trademark of AT& T Bell Laboratories.

graphical display after each modification and thus requires time proportional to the number of displayed objects. In this section we present an overview of the editing capabilities of the IDEAL Synthesizer and then describe how an attribute grammar was used to implement its incremental constraint solving algorithm.

4.1 Description

When the user invokes the IDEAL Synthesizer, two windows appear on the workstation display — a text window for inputing the IDEAL specification and a graphics window for displaying the generated pictures. The IDEAL Synthesizer is a structure editor — the user inputs the textual specification by selecting templates and either recursively replacing the "slots" with other templates or replacing the "slots" with valid textual phrases. To give the reader a feel for how the structure editor works and how the language is designed, we describe a sample editing session that creates a linked list of two items

The definitions for the objects that comprise this linked list are shown in Fig. 7 (assume that the user has already input this portion of the specification). The "var" statement defines the points that comprise each object and work variables that are used by the constraints; the constraints are self explanatory; and the "conn" statements define the lines that will be displayed. Notice that the definitions specify generic objects since not enough constraints have been provided to uniquely determine the positions of the objects' points.

We pick up the editing session at the point at which the user selects the ⟨*component*⟩ pattern and brings up a "put" template (Fig. 7).

In Fig. 8 the user has assigned a unique label to the object (head), selected the generic object "rect", and added several constraints that instantiate it with position and size information. In the graphics window a rectangle has appeared denoting this object. The remaining objects are input in a similar fashion. Fig. 9 shows the complete specification of the linked list and the corresponding graphical display.

To move the linked list to the lower left hand corner of the graphics screen, the user selects the

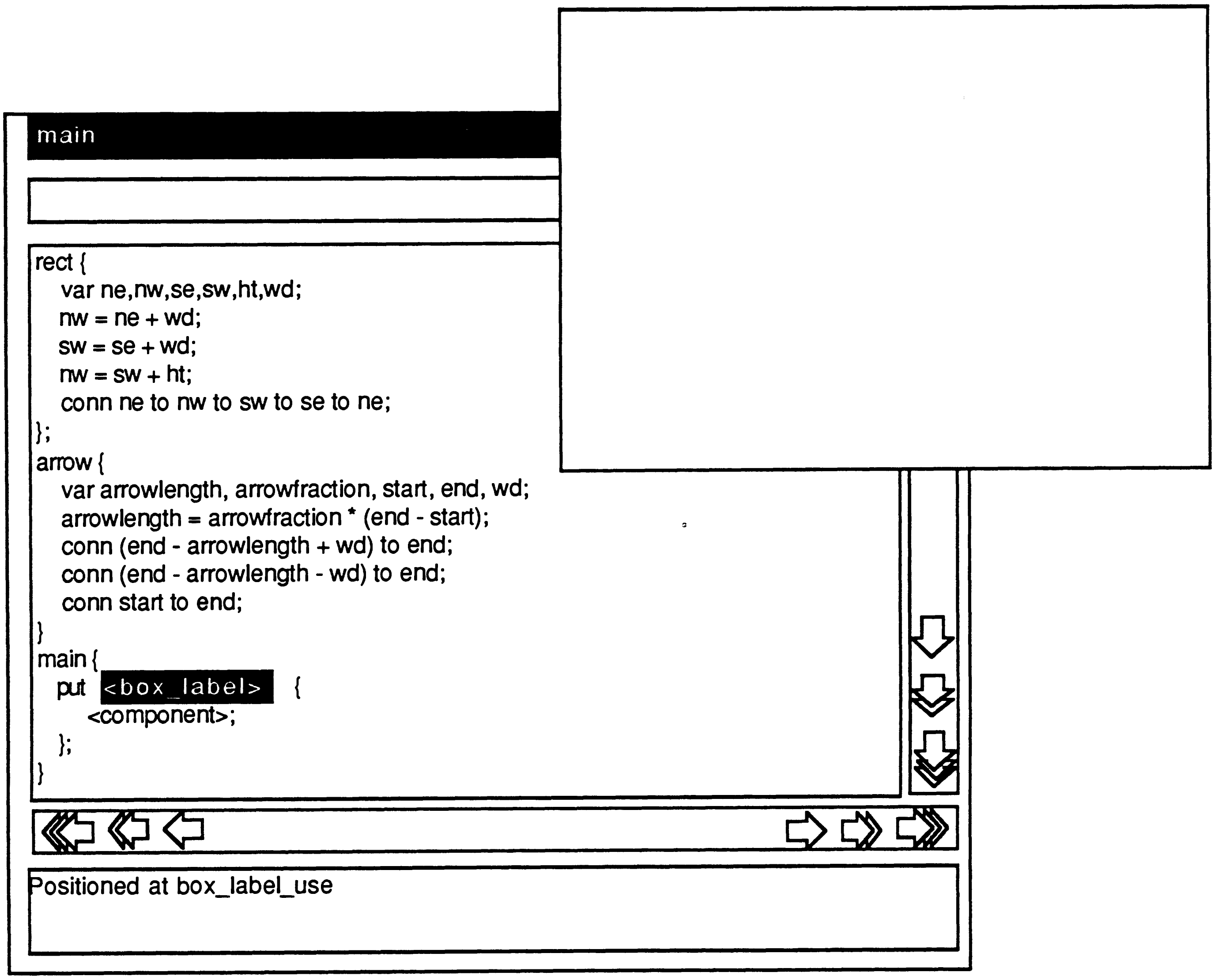

Figure 7: Definitions for linked list objects

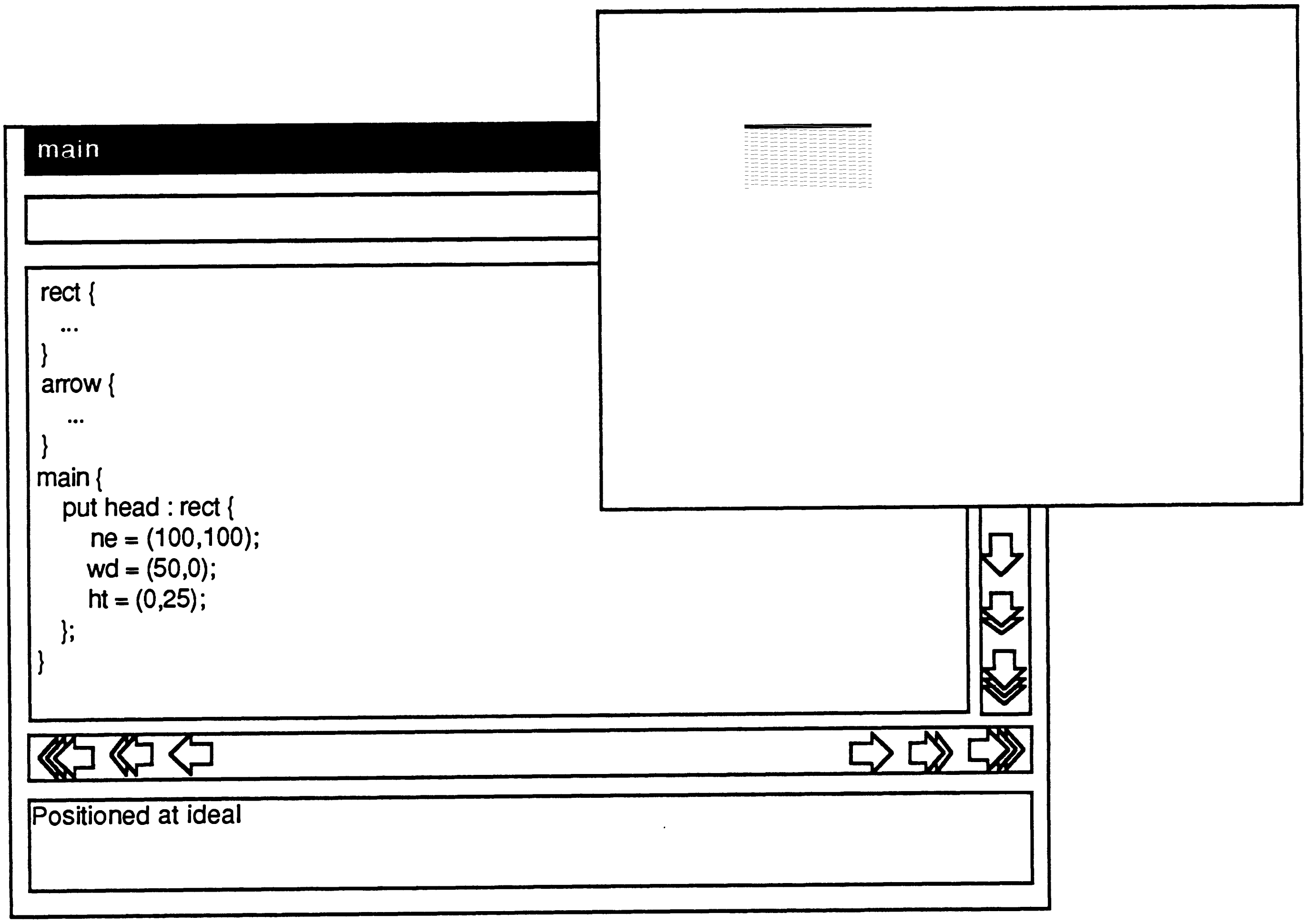

Figure 8: Specification of the first element in the linked list

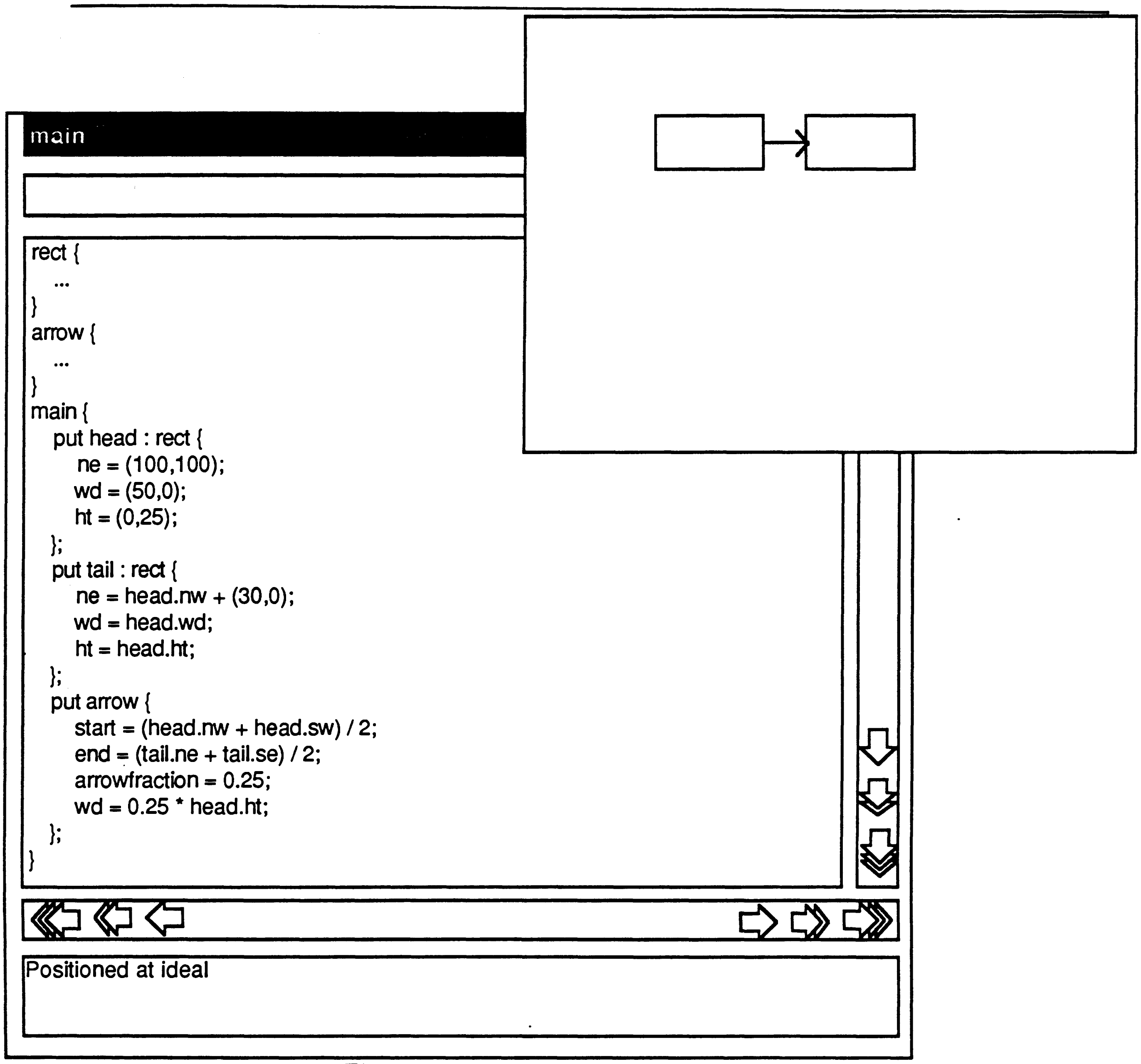

Figure 9: Completed linked list specification

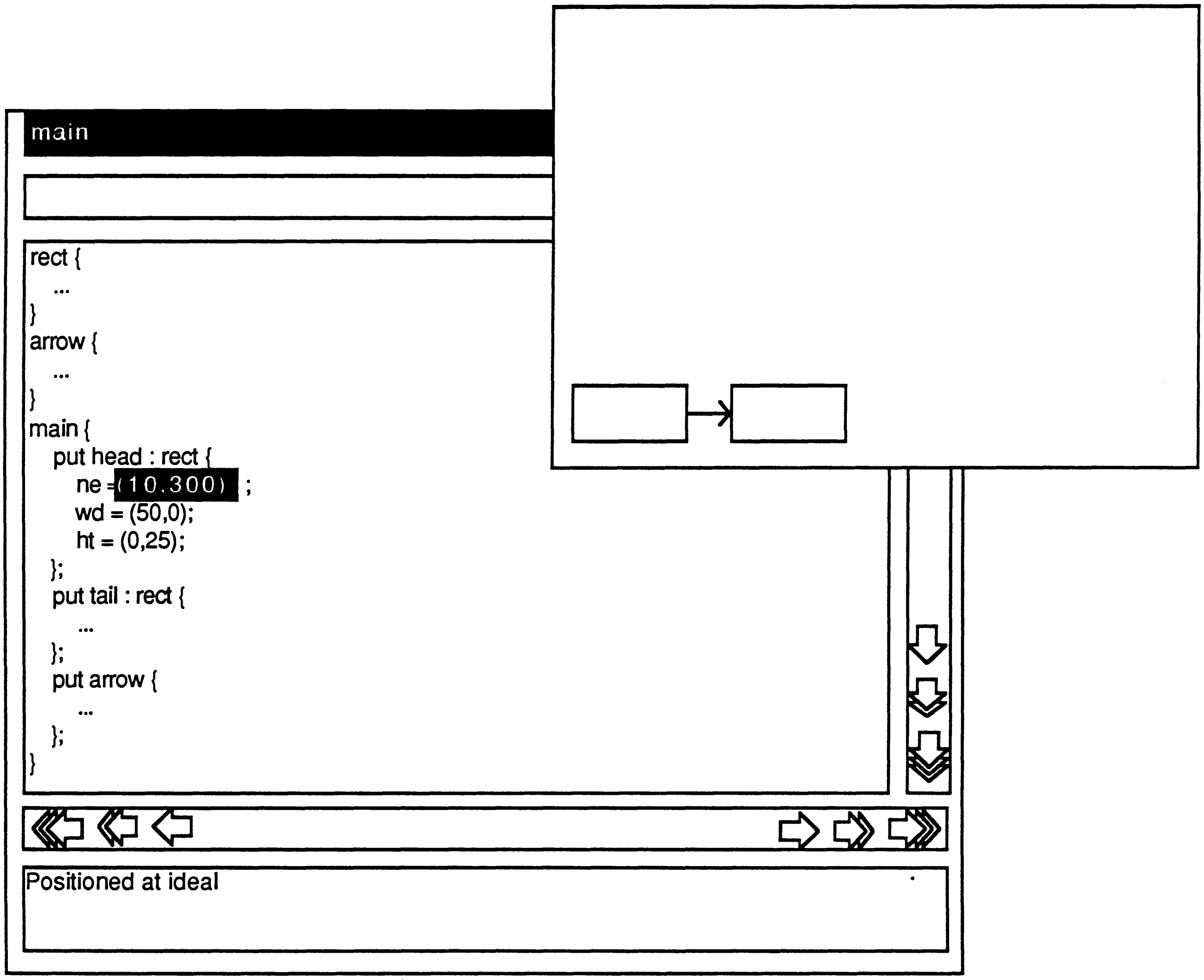

Figure 10: Specification that places the linked list in the lower left corner of the graphical display

righthand side of the constraint "ne = $(100, 100)$" and replaces it with $(10, 300)$. The graphical display is appropriately updated as shown in Fig. 10. In this case, response time is not improved by the incremental update algorithm since all three objects must be reevaluated. However, if the user moves only the second box to the bottom of the screen (i.e., if the user changes the constraint "ne = head.nw + $(30, 0)$" to "ne = head.nw + $(30, 200)$"), then only two objects, the arrow and the second box, will be reevaluated. In this instance, the response time is better than the response time that would be acheived if the entire graphical display was recomputed.

4.2 Implementation

The IDEAL Synthesizer runs under the UNIX operating system and its output can be routed to any workstation that supports XWindows. The editor represents the specification internally as an abstract syntax tree. This syntax tree is derived, of course, from the underlying attribute grammar that defines the IDEAL language. Unlike microCOSM, every editing operation in the IDEAL Synthesizer is a structural editing operation that modifies some aspect of the abstract syntax tree. For example, bringing up the put template adds a subtree associated with the "put" production to the abstract syntax tree. Similarly, changing the value of the constraint "ne = $(100, 100)$" to "ne = $(10, 300)$" causes the subtree associated with $(100, 100)$ to be replaced with the subtree associated with $(10, 300)$.

After each modification to the abstract syntax tree, some of the attributes may have changed and thus the attribute evaluator is invoked. Unlike microCOSM, the IDEAL Synthesizer makes use of Rep's incremental attribute evaluation algorithm when reevaluating attributes. The IDEAL Synthesizer uses a single abstract syntax tree that is quite large and that contains a large number of attributes; thus brute force reevaluation of the entire tree would degrade response time unacceptably. A large number of attributes is needed since attributes perform an extensive variety of functions:

1. Type checking: Symbol table and type attributes are used to ensure that there are no un-

declared variables, no undeclared objects in put statements, and no type errors. When an object definition is encountered (i.e., an object's generic definition), the IDEAL Synthesizer traverses the part of the abstract syntax tree associated with the object's definition and collects the declared variables, constraints, and lines in "environment" attributes. These environment attributes are simply lists of elements. At the root of this subtree, the environment attributes are bundled into a symbol table entry. This entry is later used by the IDEAL Synthesizer to perform the appropriate type checking when a specific instance of the object is displayed by a put statement.

2. Constraint Satisfaction: The IDEAL Synthesizer uses an incremental version of the constraint satisfaction algorithm presented in [13]. The original algorithm is an offline algorithm that collects the constraints in a specification on a queue and solves them in one fell swoop when the specification is completed. The IDEAL Synthesizer converts this algorithm to an online algorithm via attributes.

   The strategy used by the constraint solver is to simplify each constraint as much as possible when it is input, add the constraint to a queue of constraints termed the "constraint" environment, and solve the constraints when an object becomes fully specified. An object is considered fully specified when the number of constraints equals or exceeds the number of the object's declared variables.

   During the constraint simplification process, the constraint solver traverses the abstract syntax tree associated with the constraint. The abstract syntax tree shown in Fig. 11 shows a typical constraint. The constraint represented by this tree, "ne = head.nw + $(30, 0)$", has been taken from the specification of the second displayed "rect" object shown in Fig. 9.

   The constraint solver first examines the leaves of the abstract syntax tree since the equation's variables are stored in the leaves. For each variable that it encounters, the constraint solver searches the "resolved variables" environment which contains (variable, value) bindings for each variable that the constraint solver has resolved (the "resolved variables" and

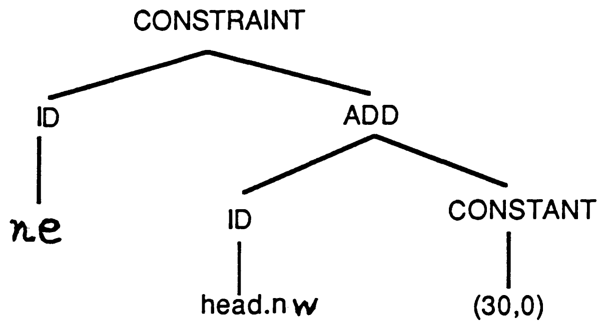

Figure 11: Abstract syntax tree for the constraint "ne = head.nw + (30, 0)"

"constraint" environments are both stored in attributes). If it locates the variable, it stores the variable's value in an attribute associated with the node. Otherwise, it stores the variable's name in that attribute. For example, in the abstract syntax tree shown in Fig. 11, ne is unresolved and thus an attribute associated with nw's ID node is given the value ne. In contrast, an attribute associated with head.nw's ID node is given the value (100, 100) since head.nw was defined in the first displayed "rect" and thus has already been resolved.

The constraint solver then proceeds to the interior nodes that specify arithmetic operations and applies these operations if their operands are constant. For example, in Fig. 11, the constraint solver examines the attributes associated with head.nw's ID node and CONSTANT's node and, finding that they are both constants, adds them and places the resulting sum, (130, 100), in an attribute associated with node ADD.

When the constraint solver reaches the node CONSTRAINT, it subtracts the right expression from the left expression and places the result in an attribute associated with CONSTRAINT. In Fig. 11, the resulting expression is SUB(ne,(130, 100)). Finally, the constraint solver proceeds to the parent of node CONSTRAINT, where it increments the attribute that is counting the number of equations by one, and adds the simplified constraint to the constraint environment. If the number of equations equals the number of the object's declared variables, the constraint solver solves the set of constraints contained in the constraint environment and adds the resulting (variable, value) bindings to the resolved variables environment.

3. Line Resolution and Display: After a set of constraints has been satisfied, the attribute evaluator extracts the expressions that define the object's lines from the symbol table and substitutes values from the resolved variables environment into the equations associated with the lines. Any lines that are resolved are stored in a picture attribute. Appropriate display routines examine these picture attributes and display the specified lines.

5 What we learned

The performances of microCOSM and the IDEAL Synthesizer cannot be directly compared since microCOSM was implemented in compiled C code whereas the IDEAL Synthesizer was implemented in the interpreted language used by the Synthesizer Generator. Nonetheless, we learned several valuable lessons by implementing our two systems and later, in comparing them. Among the lessons learned were:

1. Attribute grammars provide an elegant framework for representing constraints and extracting semantic information for constraint solvers such as initial values or the number of equations and variables.

2. When an abstract syntax tree is modified, only the object being edited and the objects being displayed should be reattributed.

3. Constraint solving can be feasibly implemented in the attributes if the power of the constraint solver is sufficiently restricted. However, factors such as the organization of the graphical display or the constraint solving algorithm may prevent the constraint solver from being

implemented in the attributes.

5.1 Attribute Grammars Provide an Elegant Framework

Both authors discovered that attribute grammars provided a natural, flexible mechanism for implementing their systems. Several features of constraint-based graphics systems lend themselves nicely to the attribute grammar formalism. First, many graphical objects have a hierarchical structure that is amenable to representation by a tree. For example, the graphical object **quad** in Fig. 4 is built up from line segments which are in turn built up from individual points. As a consequence of this hierarchical structure, the structural editing operations described in Sections 3 and 4 have a natural, formal representation in terms of subtree insertion, deletion, and replacement.

Second, the formalisms used in attribute grammars and constraint-based systems are compatible. Both define their variables applicatively in terms of constraints. Thus it is natural that an attribute grammar should be used to represent these constraints. In addition, since the constraints permitted by the IDEAL Synthesizer were amenable to incremental evaluation, it was natural to implement the IDEAL Synthesizer's constraint solver in attributes.

Thus it turns out to be quite easy to implement a constraint-based graphics system using an attribute grammar.

5.2 Attribute Evaluation Should Be Demand Driven

One of the differences between the implementation of the microCOSM and IDEAL Synthesizer editors was the way in which attribute propagation was performed. The IDEAL Synthesizer uses an eager evaluation strategy that reattributes every object in the abstract syntax tree, regardless of whether or not it is currently displayed. This strategy is consistent with one of the guiding philosophies of the Synthesizer Generator, namely, that if an object is modified, all objects that are affected by this modification should be updated immediately. In contrast, the microCOSM editor adopts a lazy propagation strategy in which objects are reattributed immediately only if they are currently displayed. Otherwise, the reattribution process is delayed until the object's next editing session or use.

Lazy evaluation is also termed demand driven propagation since attributes are reevaluated only when their values are demanded. In microCOSM only the displayed objects demand their attribute values after the abstract syntax tree is modified.

The advantage of the eager evaluation approach is that the user is immediately notified of any errors that the modification may induce in the specifications of other objects. The drawback of this approach is that it may degrade response time to an unacceptable level, since many objects that are not displayed (such as library objects) may have to be reevaluated. Thus in a graphical environment where response time is critical, lazy propagation is generally preferable.

microCOSM implements the lazy evaluation strategy by representing each definition and use of an object as a separate attributed tree in a forest of trees. When an object is edited or displayed, its entire attribute tree is reattributed. Response time is quite acceptable under this procedure since very few objects in the system are displayed at any one time. Further, when a new object is edited or displayed, the reattributing process is usually quite fast since most objects' specifications, are in general, quite short.

The efficiency of this process could have been enhanced if attribute trees were timestamped when they were attributed. Such a strategy would have allowed an object's attribute tree to be reevaluated only if one of the attribute trees on which it depended changed. In this case, a tree would be reevaluated only if its timestamp were less than the timestamp on one of the trees on which it depended. This strategy resembles a macro attribute propagation process since attributed trees are treated as functions of other attributed trees. Function reevaluations are triggered by changes to one of the function's arguments (equivalently, changes to one of the attribute trees on which the function depends) and amount to reattributing a tree (which is handled by the usual attribute propagation process). Efficiency is enhanced since entire trees can be quickly eliminated from the evaluation process by simply examining their timestamp.

Theoretically, the IDEAL Synthesizer could have implemented a similar demand driven strategy (the Synthesizer Generator supports demand driven propagation). However, a problem arises in practice. Since the IDEAL Synthesizer uses a single abstract syntax tree, it must store all generic object specifications, regardless of whether or not they are currently displayed, in the symbol table. Unfortunately, every time an entry in the symbol table changes, the symbol table demands that all objects that comprise the symbol table be reevaluated. This problem is termed the "aggregate problem". In general, the aggregate problem can arise whenever an attribute is composed of a collection or aggregate of other attributes. In the Reps algorithm, a change to one of the values that comprise this aggregate can force every value that comprises this aggregate to be reevaluated, even if they do not depend on the changed value. Readers who are interested in further details of the aggregate problem or a solution to it are encouraged to read the dissertation of Hoover [14]

5.3 Using Attribute Grammars to Implement Constraint Solving

Great care must be exercised in determining whether a constraint solver should be implemented in the attributes. In the IDEAL Synthesizer, several false starts were made before an acceptable compromise between response time, power of the constraint solver, and incremental picture generation was found. At the beginning of the project, it was thought that incrementally solving the constraints (i.e., resolving the existing set of constraints every time a new constraint was added) would reduce the variability of response time and allow portions of an object to be displayed before its specification was completed. It was expected that response time would even out since the editor would try to solve constraints as they were entered rather than in one big chunk when the specification was completed. In practice, the granularity of constraint solving was too fine: the repeated efforts at solving the constraints each time a new one was input caused response time to degrade unacceptably. Thus a coarser level of granularity was adopted whereby constraints were simplified as much as possible when they were input but were solved only after an object's specification was completed. In addition, in order to keep the response time proportional to the number of objects affected by a change to the specification, objects were prohibited from being mutually dependent. In practice, this prohibition prevents put specifications from depending on succeeding put specifications. This restriction has the effect of reducing the power of the constraint solver, and thus the flexibility of the constraint language.

If this restriction had not been imposed, the aggregate problem would have arisen and caused the work performed after a modification to be proportional to the number of displayed objects. Under the current scheme, the constraint environment that is associated with a put statement contains only the constraints from the object's generic definition plus constraints included in the put specification. Any unresolved constraints in this environment are thrown away at the root of the put statement's subtree since they cannot depend on the constraints associated with succeeding put statements and thus cannot be solved. Since each put statement starts with an empty constraint environment, the constraints associated with a put statement will only be reevaluated if 1) one of the object's constraints changes; 2) one of the put statement's constraints changes; or 3) one of the put statements on which this put statement depends changes.

If the above restriction were dropped and put specifications could depend on succeeding put specifications, then any unresolved constraints would have to be retained in the constraint environment. Since all succeeding put specifications would use this constraint environment, their constraints might have to be reevaluated if this constraint environment changed, even if the change in no way affected their constraints. This problem is a manifestation of the aggregate problem.

Even if the aggregate problem is resolved, other factors may prevent a designer from implementing constraint solving in the attributes. First, many constraint solvers, such as the one implemented in microCOSM cannot do any useful work on a set of constraints until they have all been entered. In this case, the incremental evaluation properties of attribute grammars are useless since the constraints cannot be processed until the last one is input.

Second, solving the constraints in the attributes restricts the designer's flexibility in organizing the graphical display. For example, when a constraint is deleted in microCOSM the displayed picture is not changed since the values assigned to the variables still satisfy the constraints. However, if the constraint solving is implemented in the attributes, this design decision can not be implemented, since the display is driven by attributes and the attributes will be reevaluated as soon as the constraint is deleted. These attributes will almost certainly receive different values since their computation is based on one less constraint. Thus the display will probably change.

6 Summary

This paper has described two constraint-based graphics systems, microCOSM and the IDEAL Synthesizer, and described how they were implemented using attribute grammars. Based on our implementation efforts, we concluded that attribute grammars provide an attractive framework for representing constraints and for extracting the semantic information required by the application from these constraints and the solutions to the constraints. Attribute grammars seem to be a particularly appealing mechanism for implementing constraint-based graphics systems since:

- graphical objects tend to be hierarchical objects that are amenable to representation by trees, and
- constraint-based systems and attribute grammars share the same formalism — thus constraint-based systems have a natural representation in terms of attribute grammars.

Our experiences with the two editors also taught us three valuable lessons. First, for large graphical systems that contain many non-displayed objects (for example, if there is a large library of objects), a lazy propagation strategy that only reattributes the currently edited and currently displayed objects is necessary to ensure adequate response time. Second, the decision as to whether to use the attribute grammar to implement the constraint solver depends on the tradeoffs one is willing to make among response time, power of the constraint solver, and incremental picture generation. In some cases, an acceptable compromise will be found and the attribute grammar can implement the constraint solver, as in the IDEAL Synthesizer. In other instances, the decisions made may require that the application call a separate constraint solver, as in the microCOSM editor. Third, until the aggregate problem is resolved, systems that use attribute grammars may have to have some restrictions. For example, the original IDEAL language allows objects to be mutually dependent. However, due to the aggregate problem, the IDEAL Synthesizer had to prohibit mutual dependence in order to obtain an incremental algorithm for the constraint solver.

Despite some of the drawbacks associated with the aggregate problem, we strongly recommend that designers of constraint-based graphics systems use attribute grammars to extract important semantic information from the constraints, and possibly, even to solve the constraints. We also believe that the attribute problem will soon be resolved (see for example [14]), thus allowing even more powerful constraint solvers to be implemented using attribute grammars.

Acknowledgements

microCOSM could never have been designed and implemented without the inspiration and advice of Greg Johnson, John Hopcroft, and Dean Krafft. Tim Teitelbaum and Bill Pugh both offered extensive advice and assistance during the creation of the IDEAL Synthesizer, as well as many helpful comments about drafts of this paper. We are deeply grateful for their support. We would also like to thank the members of the Synthesizer Project, whose collective effort helped make the IDEAL Synthesizer possible.

References

[1] Christopher J. Van Wyk. A graphics typesetting language. In *ACM SIGPLAN SIGOA Symposium on Text Processing*, pages 99–107, Association for Computing Machinery, 1981.

[2] Donald E. Knuth. *The Metafont Book.* Addison-Wesley, Reading, MA, 1985.

[3] Lee Alton Barford. *A Graphical, Language-Based Editor for Generic Solid Models Represented by Constraints.* PhD thesis, Cornell University, 1987.

[4] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Department of Computer Science, Cornell University, Ithaca, NY, 1985.

[5] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Conference Record of the 8th ACM Symposium on*

Principles of Programming Languages, pages 105–116, ACM, New York, 1981.

[6] A. Nico Haberman and David Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, 1986.

[7] I. E. Sutherland. *Sketchpad: A Man-Machine Graphical Communications System.* Technical Report 296, MIT Lincoln Laboratories, 1963.

[8] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3:357–387, 1981.

[9] Adele Goldberg and D. Robson. *SmallTalk 80: The Language and its Implementation.* Addison-Wesley, 1983.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, second edition, 1985.

[11] Thomas Reps. *Generating Language-Based Environments.* MIT Press, Cambridge, MA, 1984.

[12] Berc Rustem. *Projection Methods in Constrained Optimization and Applications to Optimal Policy Decisions.* Volume 31 of *Lecture Notes in Control and Information Sciences*, Springer-Verlag, New York, 1981.

[13] Emanuel Derman and Christopher J. Van Wyk. A simple equation solver and its application to financial modelling. *Software Practice and Experience*, 14(12):1169–1181, 1984.

[14] Roger Hoover. *Incremental Graph Evaluation.* PhD thesis, Cornell University, Ithaca, NY, 1987.