CONSTRAINT GRAMMARS IN USER INTERFACE MANAGEMENT SYSTEMS

Bradley T. Vander Zanden
Cornell University
Ithaca, NY 14853
(zanden@svax.cs.cornell.edu)

Abstract

Most user interface management systems provide techniques for specifying the input dialogue of an application but ignore the issue of how editing operations should change the graphical display of an application. In this paper, we discuss the CONSTRAINT system, a UIMS that incorporates new techniques for addressing the display update problem. CONSTRAINT is based on an innovative concept called constraint grammars. A constraint grammar can be used to model both the graphical display and dynamic behavior of an application. It does so by using constraints to represent relationships among parts of an application's data structures and among the application's state variables. By resatisfying the constraints after each editing operation, the application's state information and the graphical display can be appropriately updated. In CONSTRAINT, these grammars are restricted to specifying the graphical display. This restriction allows incremental, constraint-solving algorithms to be automatically extracted from a constraint grammar specification. These algorithms provide the mechanism by which the display is rapidly updated. We claim that the advantages of our approach are efficiency, conferred by the incremental algorithms, and simplicity, conferred by the declarative, modular constraints.

Keywords: Constraint Systems, User Interface Management Systems, Specification Languages

1 Introduction

User interface management sytems (UIMS's) possess considerful potential for reducing the effort required to develop and maintain direct manipulation interfaces. However, UIMS's are not widely used since they are often quite limited in the types of interfaces they can generate [12] [19] [24]. One capability that would enhance the desirability of UIMS's would be a facility for updating the display after an application's data structures are modified [10] [11] [12] [23] [24]. In many current UIMS's, the input dialogue for an application can often be coded in a matter of hours but the code that modifies the display after an application's data structures have changed may take days or even months to implement [23].

In this paper we describe the design of the CONSTRAINT system, a UIMS that introduces new techniques for addressing the display update problem. CONSTRAINT is based on a novel concept called constraint grammars. Constraint grammars use the productions of a context free grammar to represent the graphical structure of an application and constraint equations to represent the dynamic behavior of the application and the graphical relationships between parts of the application's data structures. By reevaluating the constraints after each editing operation, the application reaches a new equilibrium and the resulting values for the state and graphical variables can be used to update the display.

Constraint grammars generalize attribute grammars by interpreting numerical equations as true constraints rather than one-way constraints. For example, an attribute grammar treats the equation "ne = nw + width" as a one-way definition of the variable ne that can be solved only if the value of nw and width is known. A constraint grammar views the same equation as a two-way definition that can be solved for any of the variables. This interpretation of numerical equations gives constraint grammars considerably greater expressiveness than attribute grammars.

The CONSTRAINT system restricts constraint grammars to specifying the graphical display of an application. For example, the CONSTRAINT system can manage the graphical display of an electrical circuit but it cannot perform the computations that determine the amount of current flowing through the circuit or the voltage drops across various electrical components. CONSTRAINT imposes this restriction for two reasons. First, while the graphical display of most applications is naturally modeled in terms of numeric constraints, the dynamic behavior of many applications must be at least partially modeled by a series of procedures. Second, the equations that describe the graphical display are typically simpler than the equations that describe an application's dynamic behavior. Thus, the CONSTRAINT system is able to use a simpler constraint solver that allows the display to be more rapidly updated.

Constraint grammars have a number of advantages over alternative, procedural methods that handle the graphical display of an application. First, constraint grammar specifications require less testing and coding than their procedural counterparts. In a constraint grammar, every editing and undo operation can be handled by the same evaluation algorithm, whereas in a procedural specification, the designer must write a separate procedure to handle each such operation. In addition, the designer must test the procedural specification more extensively to ensure that each procedure works correctly both by itself and in tandem with other procedures. Second, provably correct, incremental algorithms for evaluating a constraint grammar can be automatically extracted from a constraint grammar specification. Such algorithms allow the graphical display to be incrementally updated after each editing or undo operation. In contrast, the writer of a procedural specification must manually prepare incremental update routines for each editing and undo operation.

In the remainder of this paper we will describe constraint grammars in more detail and then present an overview of the CONSTRAINT system. Particular attention will be paid to CONSTRAINT's facilities for describing the graphical presentation of the application and its facilities for specifying editing operations. More specifically, the rest of the paper is organized as follows. Section 2 describes related work; section 3 describes constraint grammars while section 4 presents an overview of the CONSTRAINT system. Section 5 examines the current status of the CONSTRAINT system and section 6 presents our conclusions.

2 Related Work

Past work on UIMS's has primarily focused on providing tools for specifying aspects of the input dialogue. Examples of such systems include Menulay [6], the Military Message System [13], Syngraph [21], GRINS [22], Squeak [7], Peridot [16] [17] [18], and vu [26]. CONSTRAINT differs from these systems in that it focuses on techniques for updating the graphical display after an application's data structures are modified.

Recently two UIMS's have been built for handling the display update problem — STUF [23] and Higgens [10] [11]. The STUF system uses procedures to control the graphical display of application objects. Each editing command has an associated procedure that walks through the application data structures, makes the appropriate changes, and performs the necessary updates to the display. The Higgens system on the other hand is based on the attribute grammar paradigm and uses equations to control the graphical display of application objects. The equations are used to model relationships among the application's objects. An editing operation may change either some of the variables in these equations or add or delete equations from the system. The graphical display is then modified in accordance with the new solution to these equations. CONSTRAINT differs from the STUF system in that it uses constraints rather than procedures to model graphical relationships and differs from the Higgens system in that it uses a constraint grammar rather than a variant of an attribute grammar.

The design of the CONSTRAINT system has been influenced by several other constraint-oriented systems including Thinglab [3] [5], IDEAL [29] [30], Juno [20], microCOSM [1] [2], and CONSTRAINTS [27]. These systems build geometric objects from primitives such as points, lines, and circles. Constraint equations can be used to represent spatial relationships between these primitives such as the fact that adjacent sides in a rectangle should be perpendicular. The user can modify the display by dragging or resizing one or more of the geometric primitives that compose an object. These actions may cause some of the constraints to be violated and thus a constraint solver must be called to reestablish these constraints. In several of these systems, constraints may also be used to model the dynamic aspects of the application. For example, Thinglab allows the user to input constraints that describe the flow of current in an electrical circuit or the effect of Hook's law on a spring. The constraint solvers used in these systems are more powerful than the constraint solver used in CONSTRAINT since they typically model the dynamic behavior of the application as well as managing the graphical display.

Finally, the internal structure of the CONSTRAINT system is modeled after the Synthesizer Generator [25]. The Synthesizer Generator is a system for transforming attribute grammar specifications into interactive, language-based editors. For example, it can take a specification of the Pascal language and create an editor that assists a user in writing a Pascal program. Similarly, the CONSTRAINT system takes a constraint grammar specification of an application's graphical structure and creates a direct, manipulation interface for interacting with this application.

3 Constraint Grammars

Constraint grammars are generalizations of attribute grammars [14]. An attribute grammar is a context free grammar with equations added to each production to calculate context sensitive information. For example, in an attribute grammar that specifies a programming language, the equations might compute type information. This information can be used to check semantic constraints, such as the requirement that the use of an identifier conform with its declared type.

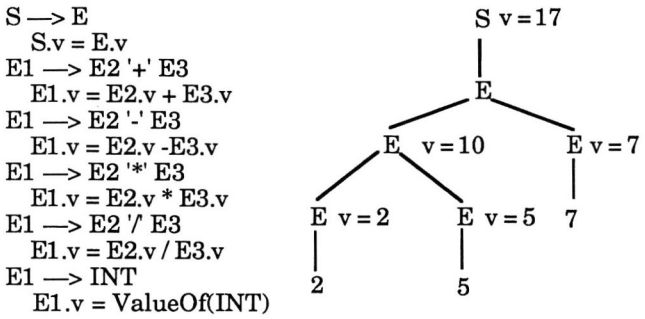

Figure 1: Sample attribute grammar for a desk calculator and an attributed abstract tree for the expression 2 * 5 + 7.

The variables in the equations are termed attributes and each attribute is owned by one of the nonterminals in the production. The attribute values represent the output of the application—assembly code for a compiler, a proof verification for a proof checker, or the locations of objects in a graphics editor. An equation may also reference any of the terminals that belong to the production (the terminals normally represent numeric or string constants). An attribute grammar treats the equations that define the attributes as one-way constraints—the attribute on the left side of the equation depends on the attributes and terminals on the right side of the equation but not vice versa. This interpretation of equations means that the values of the attributes and terminals on the right side of the equation must be known before the equation is evaluated. A sample attribute grammar for a desk calculator and one of the abstract trees that could be derived from this grammar are shown in figure 1 (an abstract tree can be thought of as a parse tree that has had syntactic details such as keywords removed).

A constraint grammar resembles an attribute grammar in that it uses a context free grammar augmented with equations and in that it can be used to derive an abstract tree that represents the structure of an application. However, it is a generalization of an attribute grammar since it treats numeric equations as true constraints rather than one-way constraints. That is, each attribute or terminal in a numeric equation depends on the other variables in the equation and thus any variable can be solved for in terms of the other variables. If necessary, a constraint grammar will perform algebraic transformations to obtain a definition for a variable as a function of the other variables in a constraint. For example, it can transform the constraint "ne = nw + wd" into the constraint "nw = ne - wd".

A consequence of their interpretation of numeric equations as true constraints is that constraint grammars may change leaves of the tree during constraint satisfaction while an attribute grammar may not. In an attribute grammar, a terminal can only occur on the right side of an equation and thus it can never be changed during the constraint satisfaction process. However, a constraint grammar will invert an equation to solve for a terminal and thus it can change the leaves of the tree. Another way of viewing this difference is that an attribute grammar derives the semantics of the application (the context-sensitive information) from the syntax of the application (the context-free information), whereas a constraint grammar allows both the semantics to be derived from the syntax and the syntax to be derived from the semantics.

A constraint grammar can be made arbitrarily more expressive than an attribute grammar by increasing the power of the constraint solver. In an attribute grammar, the numerical constraints can be no more powerful than that expressable by a lower triangular system of equations. This limitation results from the one-way interpretation attribute grammars apply to constraints. In contrast, constraint grammars permit arbitrarily complex systems of constraints, provided a sufficiently powerful constraint solver is provided. For example, constraint grammars can handle linear systems of equations that require Gaussian elimination or nonlinear systems of equations that require even more sophisticated numerical techniques.

The following two examples should help illustrate the difference in the expressiveness of attribute and constraint grammars. The first example shows how a constraint grammar can provide a much more natural, succinct definition of a problem than an attribute grammar and the second example shows how a constraint grammar can express problems that attribute grammars are incapable of expressing. In the first example, an instructor wants to construct a system for teaching students metric-English conversions. One of the subunits involves temperature conversion between degrees Fahrenheit and degrees Celcius. The teacher wishes to display two thermometers, one in °F and one in °C. If a student drags the "mercury" in one of the thermometers in one direction, the "mercury" in the other thermometer should follow suit. A sample interface for this problem is shown in figure 2. Despite the simplicity of this interface, a simple, elegant attribute grammar specification for this interface does not exist. A specification that gets the job done is:

```
temp_converter—>  FtoC_converter
                | CtoF_converter
FtoC_converter—> in_thermometer
                 out_thermometer
      out_thermometer.degrees = 5/9 X
          (in_thermometer.degrees - 32)
CtoF_converter —>  in_thermometer
                   out_thermometer
```

```
        out_thermometer.degrees = (9/5 X
              in_thermometer.degrees) + 32
    in_thermometer —> degrees: REAL
    out_thermometer —> null
```

where the declaration degrees: REAL assigns a name to the terminal REAL. The problem with this specification is that it is lengthy and prone to error since the teacher must manually invert the equation for temperature inversion. In addition, it is unnatural to distinguish between an input and an output thermometer—the teacher views the two thermometers as Fahrenheit and Celcius thermometers, not input and output thermometers. Both of these shortcomings are a direct consequence of an attribute grammar's inability to express two-way constraints.

In contrast, the constraint grammar specification for this problem is short and straightforward:

```
    temp_converter —> C: thermometer
                      F: thermometer
        F.degrees = 9/5 X C.degrees + 32
    thermometer—> degrees: REAL
```

The constraint solver will automatically invert the temperature conversion equation if the value for F.degrees is known but the value for C.degrees is not. This specification is more natural since it conforms with the teacher's physical interpretation of the problem.

The second example involves an electrical circuit system. As in the metric-English conversion example, it might be possible for an attribute grammar to provide an inelegant specification for the graphical layout of the circuit. However, since simultaneous equations are required to model the flow of current or the voltage drop across electrical components, and since an attribute grammar is restricted to one-way constraints, an attribute grammar is incapable of expressing the dynamic behavior of a circuit. In contrast, a constraint grammar with a Gaussian elimination equation solver could provide a short, elegant description of both the graphical layout and dynamic behavior of the circuit.

4 CONSTRAINT Overview

The CONSTRAINT system has a standard UIMS display architecture as shown in figure 3 [23]. As indicated by the figure, the designer provides three types of input—a constraint grammar, a graphical presentation scheme that associates graphical images with objects, and an editing dialogue that describes the actions that can be performed on the interface. These three specifications are fed through CONSTRAINT's

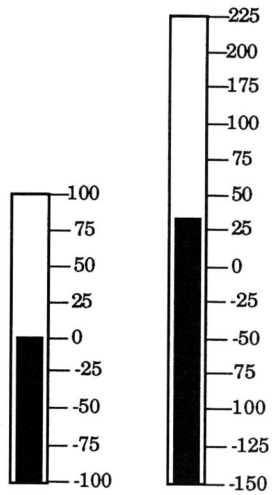

Figure 2: Interface for a temperature conversion interface

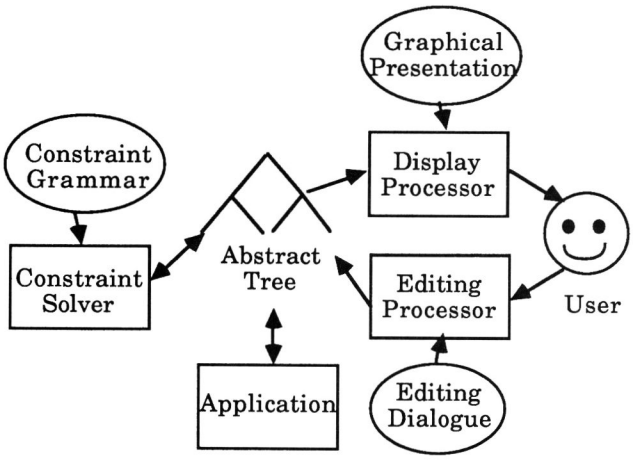

Figure 3: Architecture of the CONSTRAINT system

interface generator, which creates a working interface. The abstract tree is derived from the constraint grammar and represents the graphical structure of the application. The user interacts with the application by editing objects on the display. These actions cause the editing processor to modify the tree and to invoke the constraint solver. The constraint solver incrementally resatisfies the constraint equations and possibly modifies leaves of the abstract tree as discussed in section 3. The display processor then uses the attribute values to update the display.

The application may use the changed attribute values and the modified tree to update its data structures. As a result of these updates, the application may send messages to the editing processor asking it to modify the parts of the abstract tree that correspond to the changed data structures. Again the constraint solver is invoked to reestablish the constraints and the display processor is invoked to examine the modified attribute values and update

the display. At this point the process quiesces, awaiting further input from the user.

4.1 The Constraint Grammar

The constraint grammar component of a specification associates a set of attributes and a set of structure definitions with each object. Attributes contain information about an object's graphical layout (e.g., its position or the space it occupies) while structure definitions provide alternative ways of representing and displaying an object. For example, an electrical component might have different representations for a resistor, a capacitor, and a battery.

Each non-primitive object may be thought of as a nonterminal of the constraint grammar and each primitive object may be thought of as a terminal of the grammar. A non-primitive object corresponds to a designer defined object and a primitive object corresponds to a system provided object. There are four types of primitive objects in the CONSTRAINT system—points, bitmaps, circles and text.

A structure definition is simply a production of the grammar—the left side nonterminal associates the definition with a particular object and the right side nonterminals and terminals represent the components that comprise this version of the object. For example, the production

rect —> ne: PT nw: PT se: PT sw: PT

indicates that the non-primitive object rectangle is composed of four primitive subobjects, the points, ne, nw, se, and sw. It is permissable to give default values for these primitives. For example, initial values can be given to the endpoints of rect via the following production

rect —> ne: PT nw: PT se: PT sw: PT
(ne = (0,0); nw = (100,0); se = (0,100); sw = (100,100)).

Of course in the CONSTRAINT specification language, this production appears in a sugared form that is more palatable to the designer. However, for the sake of brevity, we will continue to use the above notation throughout the rest of the paper.

The idea of using initial values is traceable to Borning's notion of prototypes [4] and microCOSM's notion of initializers [1] [2]. The notion of building objects hierarchically from subobjects is comparable to the part-whole mechanisms used in ThingLab [3] [5] , microCOSM [1] [2], and CONSTRAINTS [27].

Each structure definition contains a set of constraints that describe relationships between the attributes associated with an object and its subcomponents. Constraints provide the mechanism by which the graphical layout of an object and its subcomponents may be computed. For example, in the above production for rect, the equations that describe the locations of rect's points might be $nw = ne + wd$, $sw = se + wd$, and $se = ne + ht$. wd and ht are attributes associated with the nonterminal rect that define rect's width and height.

Since the CONSTRAINT system is concerned with modelling the graphical display of an application and not its dynamic behavior, a powerful constraint solver such as one capable of solving simultaneous, nonlinear systems of equations is not necessary. Indeed, we have found that most of the graphical displays desired in practice can be described by a noncircular, multilinear system of constraints. The assertion that a system of equations is noncircular means that it contains no simultaneous equations, that is, the equations can be topologically ordered so that as each equation is enumerated, it depends only on the equations that have preceded it. The assertion that a system is multilinear means that each equation is linear in each of its variables separately. That is, for $1 \le i \le n$, $f(x_1,\ldots,ax_i + b,\ldots,x_n) = af(x_1,\ldots,x_i,\ldots,x_n) + f(x_1,\ldots,b,\ldots,x_n)$ [8]. Thus a permissable set of equations would be $z = xy$, $y = x/3$, $x = 12$ and an impermissable set of equations would be $a + b = 10$, $a - b = 6$ or $x^2 + y^2 = 9$.

A technique known as "propagating degrees of freedom" can be used to solve a noncircular, multilinear system of equations in $O(n^2)$ time where n is the number of variables [3] [15] [28]. In a constraint grammar, the time drops to O(n) since each variable can belong to only a constant number of equations. When a set of variables changes, the time required by this algorithm to reestablish the constraints is O(INFLUENCED) where INFLUENCED is the number of equations directly or indirectly affected by the changed variables. The implementation of this technique is more fully described in a separate paper.

4.2 The CONSTRAINT Editor

The CONSTRAINT system permits two types of editing operations to modify the abstract tree—**dragging operations** that resize or reposition existing objects and **structural editing operations** that delete, add, or replace objects. Dragging operations change the values of primitive objects such as points. As such they change leaves of the abstract tree. Structural editing operations change nonprimitive components and thus change interior nodes of the tree.

A dragging operation is handled in the following manner. When the user selects a primitive and starts to drag it, the CONSTRAINT editor identifies

the affected leaf of the abstract tree and changes the value of this leaf so that it corresponds with the new value of the primitive. The constraint solver is then called to reestablish the constraints that are violated by this change. This process is repeated as long as the user keeps the mouse button depressed and is changing the picture. If the constraint solving can be performed fast enough, the display will change as the mouse moves, providing the user with valuable feedback about the changes that are induced by modifying the object [1] [2]. As noted later in the paper, this objective is typically achieved.

Structural editing operations are handled in much the same manner as dragging operations. When the user selects an object and adds or deletes a component, the editing processor must find the affected area in the abstract tree and either add or delete a subtree from it. In doing so, the editor will almost surely make some of constraints at the point of subtree insertion or deletion inconsistent and thus the constraint solver must be called to reestablish the constraints.

Structural editing operations are accomplished via transformations [25]. A transformation is a function that maps a set of subtrees that satisfy a selection pattern into a new set of subtrees. The syntax of a transformation is

transform <selection pattern> on "command name"
{ <actions> }

As the user selects objects on the screen, the editor finds the tree nodes associated with the selected objects and attempts to match them against the selection pattern of each transformation. Those transformations for which the matching process succeeds have their command names added to a menu of valid transformations. When the user selected a command name from the menu, the specified actions from the appropriate transformation are performed and the display is updated. Actions may include deleting, creating, or swapping subtrees.

For example, returning to the metric-English conversion example, the teacher may want to allow a student to choose either a temperature conversion application or a distance conversion application. Thus the teacher could include as part of the specification, the productions:

```
metric_English —> initial_interface
                | temp_converter
                | distance_converter
distance_converter —> inches: ruler  cm: ruler
   inches.distance = cm.distance / cm_to_inches
   cm_to_inches = 2.54
ruler —> distance: REAL
```

and the transformations:

```
transform on "metric-english lesson" {
   create(metric_English);
}
transform metric_English on "temperature unit"
{
   create(temp_converter);
}
transform metric_English on "distance unit" {
   create(distance_converter);
}
```

The first transformation allows the user to boot the metric_English interface. Since no objects initially exist, this transformation is valid when no objects have been selected. When the metric_English interface is booted by the user, the CONSTRAINT system will build the object initial_interface since the first definition of an object is always the default definition. If the student selects the object initial_interface and then the command "temperature unit", the subtree rooted at the node default_interface will be replaced by a subtree rooted at a node temp_converter. The graphical display for the temperature converter interface will appear and the student will be able to proceed with the unit.

The interaction techniques permitted in the initial implementation of the CONSTRAINT system are limited to mice and menus. These techniques are more than adequate for exercising the features of the CONSTRAINT system. The range of interactive techniques might be increased in the future, perhaps via one of the UIMS's mentioned in section 2 on related work. Constraint grammars could also be used to provide additional interactive devices such as buttons, valuators, and slider. The techniques for defining such devices using attribute grammars have been implemented in the GRINS system [22] and they could be easily extended to constraint grammars. However, since the CONSTRAINT system is supposed to explore issues relating to the management of an application's graphical display, not interactive techniques, designing an elaborate set of interactive techniques is not a priority item.

4.3 Graphical Presentation

To convert the abstract tree into a graphical display, designers need a set of drawing commands that they can associate with each node of the abstract tree. The graphics commands incorporated in CONSTRAINT are drawn primarily from a subset of the commands permitted in Van Wyk's IDEAL picture creation language [29] [30]. These commands were chosen since they are device independent and "pen" independent. By "pen" independent we mean that the designer does not have to worry about moving the drawing pen to the appropriate place on the screen before drawing commences. Instead, the designer, through the use of attributes and terminals, simply indicates where on the display an

```
temp_converter —> C: thermometer  F: thermometer
     C.scale_start = -100;
     C.scale_end = 100;
     C.scale_incr = 25;
     F.scale_start = -150;
     F.scale_end = 225;
     F.scale_incr = C.scale_incr;
     C.se = (100,0);
     C.wd = (10,0);
     C.ht = (0,200);
     F.se = C.se + C.wd + (20,0);
     F.wd = C.wd;
     F.ht = (C.ht X (F.scale_end - F.scale_start))
                / (C.scale_end - C.scale_start);

/*  graphical image for a rectangle */
rect —> ne: PT  nw: PT  se: PT  sw: PT
     polygon(color,{ne,nw,sw,se});

/* graphical image for the outline of a thermometer */
thermometer —> degrees: REAL
     put outline: rect {
          se = thermometer.se;
          wd = thermometer.wd;
          ht = thermometer.ht;
          color = "opaque";
     }

/* graphical image for the mercury */
     put mercury: rect {
          se = outline.se + (2,0);
          wd = outline.wd - (4,0);
          ht = (0,thermometer.degrees);
          color = "red";
     }

/* graphical image for a thermometer scale */
numbered_line_seg —> start: PT  end: PT  label: INT
     conn start to end;
     label at { text.nw = end + (3,0); }

/* graphical image a numbered thermometer scale */
thermometer —> degrees: REAL
   conn outline.se to outline.ne using
     (scale_end-scale_start)/scale_incr + 1
     numbered_line_seg {
        start = x;
        end = x + (3,0);
        label = thermometer.scale_start + (i-1)*
                     thermometer.scale_incr
     }<x,y,i>
```

Figure 4: Specification of the graphical presentation for the temperature conversion example

image should be located and the graphics commands automatically ensure that the drawing pen is moved to the appropriate position.

A sample specification for the graphical presentation of the temperature conversion interface is given in figure 4. The "conn" command draws lines through the specified points while the "polygon" command draws a polygon through the named points. The interior of the polygon is filled with the color contained in the "color" variable. The *put* command places an object such as rect at the location defined by *equations*. The object may be optionally identified by a label, such as "outline" or "mercury". Any primitive or nonprimitive object may be used by a put command. If a nonprimitive object (i.e., a nonterminal) is specified, the drawing commands associated with this nonprimitive will be used to draw the object. In this case, the equations must also define all of the parameters used by these graphics commands. These equations are true constraints with a two way interpretation.

The command "label at { text.nw = end + 3; }" is a string drawing command that causes the value of the variable label to be printed left justified inside the text box whose boundaries are defined by the equation inside the brackets. The default width of a text box is the width of the string. For longer strings the designer can specify a width for the box, in which case the string will be broken into lines to fit inside the box.

Finally the "conn using" command is a generalization of the conn command. It draws an object, in this case the object numbered_line_seg, n times along the imaginary line from outline.se to outline.ne where n is equal to "(scale_end-scale_start)/scale_incr + 1". The object may be a primitive or nonprimitive object and the *equations* assign values to the object's attributes and primitives so that the object's constraints can be solved. The x, y , and i variables are used to specify intermediate points along the line and may be used as variables by the equations. The "conn using" command is actually an iterative construct and is equivalent to the expression:

```
for i = 1 to n {
     put object_name {
          x = ((i-1)/n)[pt1,pt2];
          y = (i/n)[pt1,pt2];
          equations
     }
}
```

where a[pt1,pt2] is shorthand for pt1 + a(pt2 - pt1) [29] [30]. Since CONSTRAINT's graphical language does not include the for statement, "conn using" must be used instead.

5 Current Status

A prototype of the CONSTRAINT system has been implemented that runs under the UNIX[1] operating system and uses the XWindows graphics package. The current system consists of 13,000 lines

of C code and incorporates most of the features of the CONSTRAINT system described in this paper. In particular, mechanisms for specifying the constraint grammar and the transformations are completely in place. The conn and put display commands have been completely implemented and the polygon command with the exception of color specification is working. Points are the only primitive object currently supported by the prototype.

Initial experience with the prototype has been good. The constraint solver incrementally finds a new evaluation sequence after each transformation or drag operation rather than recomputing the sequence from scratch. In practice, this means that the response time is proportional to the number of constraints directly or indirectly influenced by a change rather than to the total number of constraints in the system. Thus operations that involve relatively small changes to the display, such as swapping two small subtrees in a binary tree application, achieve almost instantaneous response times of a second or so. We have also found that objects being dragged around the screen approach mouse speed (i.e., follow the mouse cursor around the screen without too much lag time) in cases that involve up to 1000 reevaluated constraints.

6 Conclusion

This paper has introduced a new concept, constraint grammars, and shown how they are incorporated in CONSTRAINT, a user interface management system. Constraint grammars provide a promising mechanism for modeling both the dynamic aspects and the graphical display of an application. The CONSTRAINT system explores how they might be used to manage an application's graphical display. Constraint grammars fill this role by providing a part-whole mechanism for defining the structure of objects and a constraint mechanism for describing graphical relationships between objects. A constraint solver that incrementally resatisfies the constraints after each editing operation allows the display to be rapidly updated. A great many user interfaces can be represented in this scheme including computer-aided drafting [15], layout systems [9], graphical programming systems [3] [5], document preparation facilities [29], education systems, and business applications such as histograms, pie charts, and task scheduling charts [9]. As such constraint grammars provide a natural, flexible mechanism for specifying the graphical display of an application and for reducing the possibility for error in a specification.

Acknowledgements

The CONSTRAINT system would not have been possible without the helpful input that Dexter Kozen, Tim Teitelbaum, and Bill Pugh have provided throughout the design of the CONSTRAINT system. In addition, comments provided by John Fields, Roger Hoover, Chet Murthy, and Steve Peckham helped improve the presentation of this paper.

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

References

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

[2] Barford, L.A. and Vander Zanden, B.T. 1987. *Attribute Grammars in Constraint-based Graphics Systems*, Technical Report 87-838, Cornell University, June 1987.

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

[4] Borning, A.H. 1986. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, Nov. 1986.

[5] Borning, A. and Duisberg. 1986. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5:345-374.

[6] Buxton, W., Lamb, M.R., Sherman, D. and Smith, K.C. 1983. Towards a comprehensive user interface management system. In *SIGGRAPH'83 Conference Proceedings*, pages 35-42, ACM, New York.

[7] Cardelli, L. and Pike, R. 1985. Squeak: a language for communicating with mice. In *SIGGRAPH'85 Conference Proceedings*, pages 199-204, ACM, New York.

[8] Edwards, C.H., Jr. 1973. *Advanced Calculus of Several Variables.* Academic Press, New York.

[9] Foley, J.D. and Van Dam, A. 1982. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, MA.

[10] Hudson, S.E. 1986. *A User Interface Management System Which Supports Direct Manipulation.* PhD thesis, University of Colorado.

[11] Hudson, S.E. 1986. Implementing a user interface as a system of attributes. In *2nd ACM SIGSOFT/SIGPLAN symposium on practical software development environments*, pages 143-149.

[12] Hudson, S.E. 1987. UIMS support for direct manipulation interfaces. *Computer Graphics*, 21(2):120-124.

[13] Jacob, R.J.K. 1983. Using formal specifications in the design of a human-computer interface. *Communications of the ACM* 26,4 (April), 259-264.

[14] Knuth, D.E. 1968. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127-145.

[15] Leler, W. 1988. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, Reading, MA.

[16] Myers, B.A. and Buxton, W. 1986. Creating Dynamic Interaction Techniques by Demonstration. In *SIGGRAPH'86 Conference Proceedings*, pages 249-258, ACM, New York.

[17] Myers, B.A. 1987. Creating Dynamic Interaction Techniques by Demonstration. In *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems*, pages 271-278.

[18] Myers, B.A. 1987. *Creating User Interfaces by Demonstration.* PhD thesis, Toronto University, Toronto, Canada, May 1987.

[19] Myers, B.A. 1987. Gaining general acceptance for UIMSs. *Computer Graphics*, 21(2):130-134.

[20] Nelson, G. 1985. Juno, a Constraint-Based Graphics System. In *SIGGRAPH'85 Conference Proceedings*, pages 235-243, ACM, New York.

[21] Olsen, D.R. and Dempsey, E.P. 1983. SYNGRAPH: a graphical user interface generator. In *SIGGRAPH'83 Proceedings*, pages 43-50, ACM, New York.

[22] Olsen, D.R., Dempsey, E.P., and Rogge, R. 1985. Input/Output linkage in a user interface management system. In *SIGGRAPH'85 Proceedings*, pages 191-197, ACM, New York.

[23] Olsen, D.R. 1986. Editing templates: a user interface generation tool. *IEEE Computer Graphics and Applications*, 6(11):40-45.

[24] Olsen, D.R. 1987. Larger issues in user interface management. *Computer Graphics*, 21(2):134-137.

[25] Reps, T. and Teitelbaum, T. 1987. *The Synthesizer Generator Reference Manual.* Department of Computer Science, Cornell University, Ithaca, NY.

[26] Singh, G. and Green, M. 1987. Visual Programming of Graphical User Interfaces. In *Proceedings of the 1987 Workshop on Visual Languages*, pages 161-173.

[27] Sussman, G.J. and Steele, G.L., Jr. 1980. CONSTRAINTS—A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 14:1-39.

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

[29] Van Wyk, C.J. 1981. IDEAL User's Manual. *Bell Laboratories*, Murray Hill, NJ.

[30] Van Wyk, C.J. 1982. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163-182.