CONSTRAINT GRAMMARS—A NEW MODEL FOR SPECIFYING GRAPHICAL APPLICATIONS

Bradley T. Vander Zanden

Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
bvz@a.gp.cs.cmu.edu

ABSTRACT

User Interface Management Systems often attempt to separate the graphical and nongraphical aspects of an application, but rarely succeed. *Constraint grammars* provide a new model for specifying interfaces that achieves this goal by encapsulating the data structures in a single package, and providing a powerful transformation-based editing model for manipulating them. Constraint grammars incorporate a number of important tools, such as the *part-whole hierarchy*, *almost hierarchical structures*, and *multidirectional constraints*, that permit designers to specify a wide variety of graphical applications, including simulation systems, program visualization systems, and visual programming environments.

KEYWORDS: Constraint Systems, User Interface Management Systems, Specification Languages, Graphical Interfaces, Encapsulation, Programming Environments

INTRODUCTION

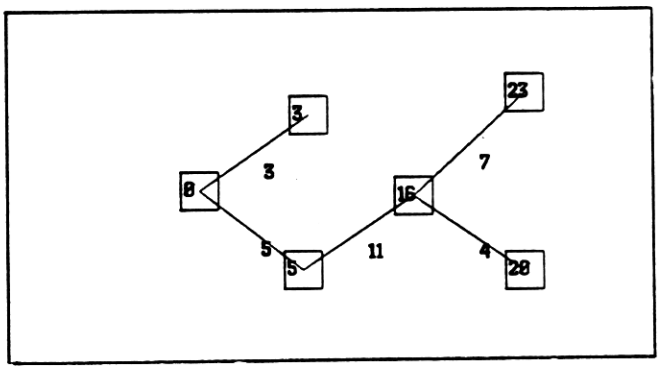
User Interface Management Systems (UIMS's) typically strive to separate the nongraphical aspects of an application from the graphical aspects of its user interface. The rationale behind this separation is that the application programmer should focus on the nongraphical aspects of the application and the interface designer should focus on the graphical aspects. However, when the application must display complex data structures, as in program visualization systems, visual programming languages, visual programming environments, and games, this separation actually shifts the burden of updating the display to the application programmer. The reason is that the graphics system knows nothing about the data structures being manipulated by the application. All it can do is dumbly respond to a stream of output tokens sent by the application that direct it where to position objects on the display.

This paper proposes a new graphical model, *constraint grammars*, that integrates the application's and graphic's data structures. This integration actually frees the application programmer from worrying about the graphical aspects of the interface, shifting the responsibility back to the interface designer where it belongs. Thus it achieves the goal that UIMS's so often seek but fail to obtain.

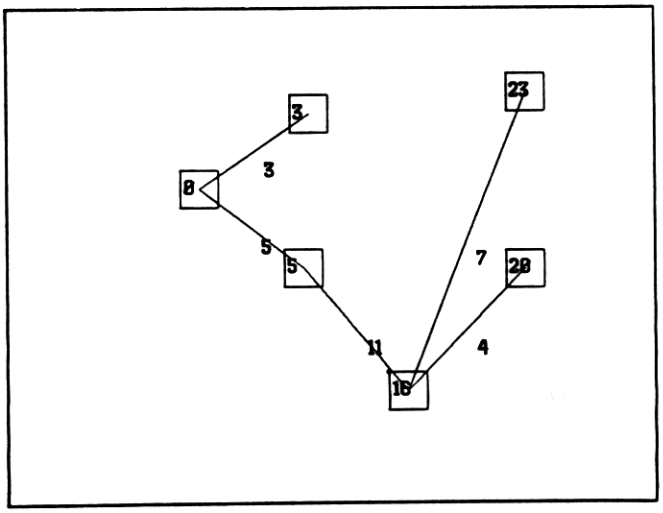
Constraint grammars integrate ideas from constraint-based, simulation systems and programming environments to produce a powerful model that is capable of specifying not just simulation systems and textual programming environments, but also visual applications, such as the ones listed above, that manipulate complex data structures. Constraint grammars use the productions and attributes of an attribute grammar to represent the data structures of an application and constraint equations to represent the dynamic, graphical behavior of the application. It goes beyond previously proposed constraint-based graphics paradigms in that it contains a powerful transformation-based editing model that permits the manipulation of complex data structures, such as lists, trees, and sets. From both the interface designer's viewpoint and the application programmer's viewpoint, the data structures are encapsulated in a package, the transformations are a set of messages or procedures that manipulate these data structures, and the attributes are the exported portion of the data structures that can be related via constraints. Thus, once the application programmer and interface designer agree upon a common set of data structures and an appropriate set of transformations, they can perform their tasks separately.

A graphical interface specified and implemented using constraint grammars will work as follows. When a transformation is invoked, the encapsulated data structures will be updated and a constraint solver will be called to reestablish the constraints. The application may then use the updated attribute values to perform additional processing, possibly modifying the data structures via transformations in the process and causing the constraint solver to be reinvoked. Once the application has finished its processing and the constraint solver has resatisfied all constraints, the graphics attributes can be used to redraw the display.

Constraint grammars have been implemented in CONSTRAINT, a system that automatically generates a



(a)



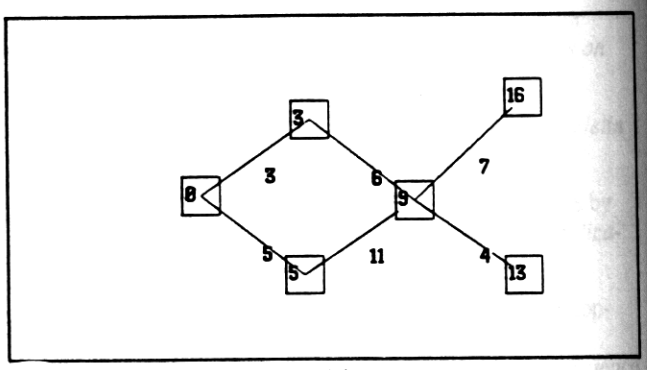
(c)

Figure 1: Display for the shortest path problem.

mouse and menu-based version of an application from its constraint grammar specification [14,15]. CONSTRAINT runs on top of the X window package and incorporates a novel incremental constraint satisfaction algorithm that provides instantaneous real-time response to editing operations, even those that affect hundreds of constraints. Thus far CONSTRAINT has been used to create a number of graphical applications, including:

1. graphical input devices such as analog and digital gauges;
2. a physics experiment that demonstrates the effects of force on a screwplate;
3. mathematics experiments that visually demonstrate geometric theorems;
4. a visual representation of the shortest path problem; and
5. a system for visually representing and manipulating binary trees.

A preliminary version of constraint grammars and the CONSTRAINT system was presented in [14]. This paper expands the constraint grammar model and describes it in greater detail.



(b)

SAMPLE APPLICATIONS

Figures 1 and 2 illustrate two applications that have been implemented using CONSTRAINT. Figure 1 shows a display for the shortest path problem. The labels on the edges denote the distance between two nodes and the number inside each node represents the shortest path from a designated node (the leftmost node) to that node. Adding or deleting an arc (Figures 1.b) causes the shortest path solutions to be automatically updated. The user can also alter the layout of the graph (Figure 1.c) by grabbing a corner of a node with the mouse and dragging it across the screen. The adjacent arcs, their labels, and the node's cost are all moved as well.

Figure 2 shows a sample application involving binary trees. This application allows the user to create or delete trees, add or delete children, split or join trees, swap children or subtrees, and move or scale nodes of a tree. Figure 2 shows several operations performed by a user including swapping two children, splitting a tree, and scaling a tree.

These two applications illustrate the weaknesses inherent in two alternative interface specification models—object systems and attribute grammars. Object systems do not have the powerful editing models that are required to manipulate the binary trees in Figure 2, while attribute grammars do not allow the almost hierarchical structures or multidirectional constraints that are needed to implement the shortest path problem in Figure 1. Almost hierarchical structures permit objects to share common components, while multidirectional constraints permit an equation to be satisfied by modifying any of its variables.

RELATED WORK

Relatively few general purpose UIMS's emphasize the presentation of application data. Most UIMS's provide an "escape hatch" to the application procedures such as *active values* that link display variables with important variables in the application's data structures [7,12]. However, these approaches normally shift the burden of updating the graphical display to the application.

Consequently, a number of researchers have begun investigating an approach in which the interface and application are integrated and share common data structures. Such an approach allows the interface to browse the application's data structures, effectively relieving the

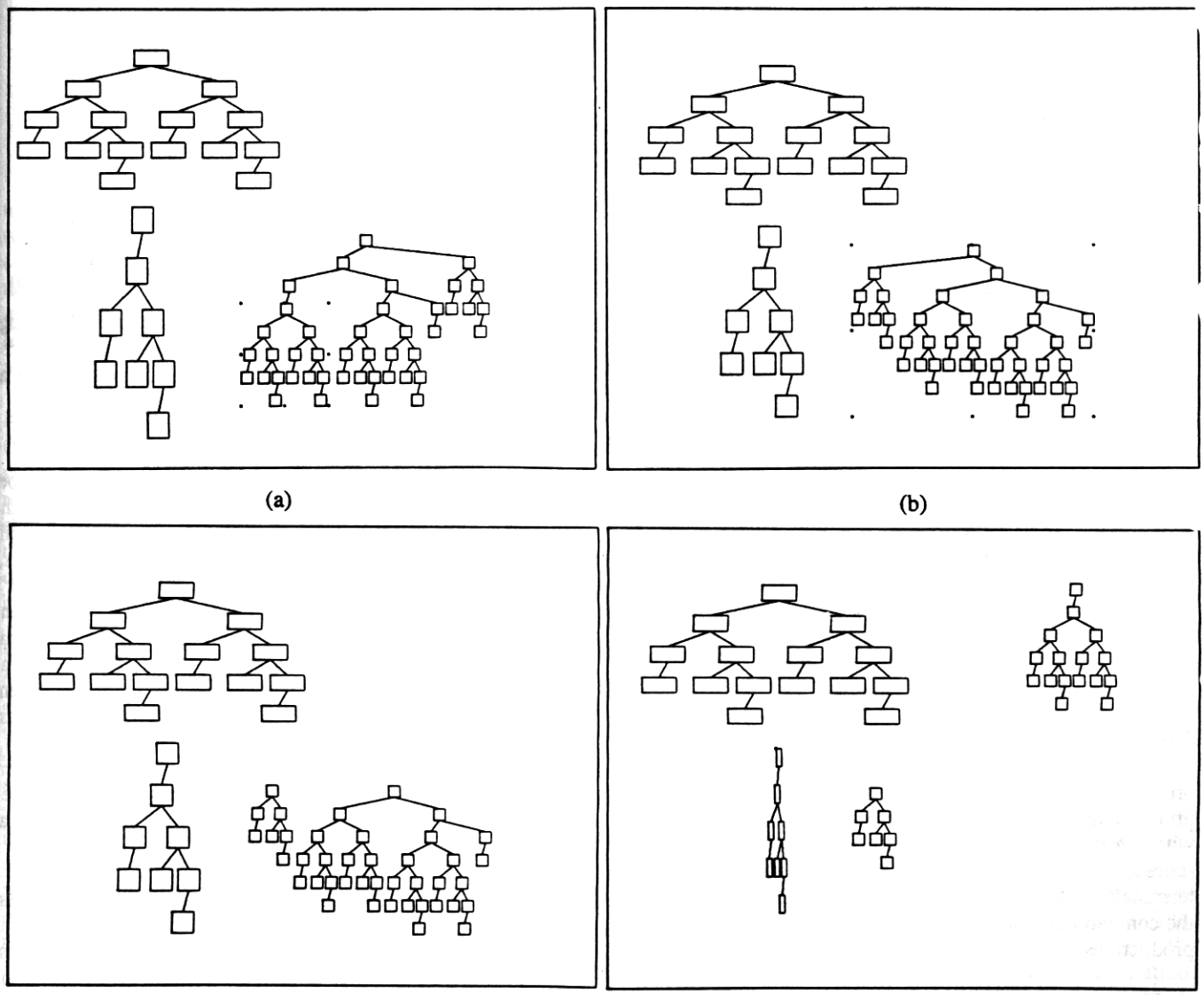


Figure 2: Display for a binary tree application.

application programmer of any need to consider the problem of updating the display. STUF [9], Higgens [4,5], and GROW [2] are examples of systems that implement this approach. Of these systems, only STUF provides an editing model that is truly capable of manipulating complex data structures. However, it takes a procedural approach, whereas constraint grammars emphasize a declarative approach.

Constraint grammars have also been influenced by the ideas found in simulation systems, such as ThingLab [3], CONSTRAINTS [13], and microCOSM [1]. These systems provide constraints that allow the designer to specify both the display and semantic properties of an application, thus allowing the simulation of physical systems such as electrical circuits or springs. They also provide tools, such as the part-whole hierarchy (described in the next section) that effectively utilize these constraints in building simulations.

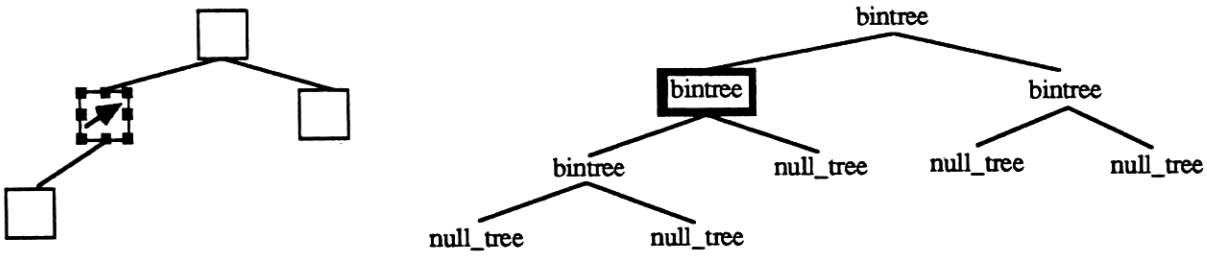
Finally constraint grammars have integrated the powerful transformational editing models that are found in the literature on attribute grammars. The Synthesizer Generator [11] is an example of a nongraphical UIMS that incorporates transformations to help designers build programming environments for handling textual objects such as programs or theorems.

CONSTRAINT GRAMMARS

Constraint grammars incorporate a number of ideas that have been found to be particularly useful in graphical settings. These ideas include the *part-whole hierarchy* (also known as the *principle of compositionality*), *almost hierarchical structures*, and *multidirectional constraints*.

Part-Whole Hierarchy

In a part-whole hierarchy, objects are built from collections of subparts, where the subparts may be either previously defined objects or primitive objects such as points, text, and bitmaps. For example, a thermometer may consist of two rectangles representing the mercury and outer shell and a



(a) (b)

Figure 3: A binary tree as it is externally presented to the user (a) and internally presented to the application and interface (b)

labeled line segment corresponding to the scale. This hierarchical decomposition of an object reduces the complexity of specifying an application by allowing the designer to assemble new objects from existing parts without having to repeat the work of respecifying these parts from scratch.

Constraint grammars provide the part-whole hierarchy by adopting the attribute grammar framework. An attribute grammar is a context-free grammar with equations added to each production to calculate context-sensitive information, such as type information [6]. The variables in the equations are called *attributes*, and each equation computes the value of one attribute. Each attribute is owned by one of the nonterminals that form the production.

The nonterminals in a constraint grammar correspond to designer-defined objects, while the terminals represent primitive objects such as points, text, and bitmaps. The productions of the constraint grammar define an object, with the nonterminal on the left side of the production representing the object's name and the nonterminals and terminals on the right side of the production representing the components that comprise the object. For example, the productions

```
bintree —> left_child : bintree
           right_child : bintree
```

and

```
bintree —> null_tree
```

indicate that a binary tree has two representations—a displayed representation, and a null, nondisplayed representation. The displayed representation consists of two components, a left child and a right child, both of which are also binary trees, while the null representation consists of one component, a null tree. Figure 3 illustrates how these productions are used to build an internal representation of the application. Figure 3a shows a binary tree as it is externally presented to the user, and Figure 3b shows the binary tree as it is internally represented. Notice how the internal representation is composed from the two productions for a bintree.

The attributes associated with each nonterminal maintain display information about an object's layout, such as its size and dimensions, and semantic information about the object's behavior, such as the amount of current flowing through an electrical component, the amount of force applied to a spring, or the allocation of resources in a production system.

Almost Hierarchical Structures

Many applications that arise in practice cannot be adequately represented by a hierarchical structure in which there are no shared objects. For example, electrical components need to share common terminals. Thus, simulation systems typically introduce the idea of "merges" in which two data structures representing the same common object are conceptually combined into one common data structure. For example, if a resistor is connected to a battery, the two terminals that form the connection between these components are merged into one terminal. Merges give rise to *almost hierarchical* structures [13]. Conceptually, such models are very powerful since the user can specify a relationship between any two objects on the screen, provided that one object can be legitimately made a subcomponent of the second object.

Constraint grammars provide almost hierarchical structures by allowing the internal representation of a graphical application to be a directed graph rather than a tree (cycles are permitted). In other words, the information extracted from an application's specification using the grammar's productions can be represented as a directed graph. Thus an instance of a nonterminal may have multiple parents. Each node of the graph corresponds to an instance of the left nonterminal of a production and its successors correspond to instances of the nonterminals and terminals on the right side of the production. The leaf nodes of the graph are terminals, and the interior nodes are nonterminals. As in an attribute grammar, each node N that represents an instance of the nonterminal X contains a set of attribute instances that correspond to the attributes of X.

Multidirectional Constraints

The part-whole hierarchy and merges provides ways of relating the structure of objects. Multidirectional constraints, on the other hand, provide a way of relating the attributes of these structures, such as their height, their width, their position on the screen, and the amount of space they occupy. A constraint specifies a relationship between the attributes of two or more subparts that must always be satisfied. These relationships might describe a syntactic property of an application such as the alignment of two objects, or a semantic property of an application such as the amount of current flowing through a circuit. For example

the binary tree specification uses the constraints

left_son.top = bottom - space/2 + left_son.space/2 + nodeheight

right_son.top = bottom + space/2 - right_son.space /2 + nodeheight

to position the left and right children of a node on a screen, so that the binary trees rooted at these children do not overlap. In this example, the attributes *top* and *bottom* refer to the top and bottom of a binary tree node, *space* refers to the space occupied by the binary tree rooted at a binary tree node, and *nodeheight* refers to the height of the node.

AN EDITING MODEL

One of the factors that has limited systems that utilize part-whole hierarchies and almost hierarchical structures to the realm of graphical simulations and picture drawing is the absence of a powerful editing model that permits users to manipulate the part-whole hierarchy. Merges and constraints provide a limited capability to modify objects, but for many graphical applications, such as the binary tree example in Figure 2, these capabilities are too limited. Merges typically can only combine points or lines. Constraints can manipulate the attributes of a structure, such as requiring two resistances to be the same. Neither of these concepts allows the user to conveniently modify the structure of the more complex objects that can be created within the part-whole hierarchy framework, such as sets, trees, and lists. For example, the binary tree system shown in Figure 2 does not permit a user to add a child to a tree node if such a child already exists. However, neither a merge nor a constraint allows the application or interface to examine the tree to determine if such a condition exists. In this section, we present an editing model that is powerful enough to fully exploit the richness offered by constraints and the part-whole hierarchy.

The editing model we propose for constraint grammars is based on *transformations* [11]. A transformation is a function that maps a collection of nodes in the directed graph and a command into a modified set of nodes. A transformation consists of a selection pattern, a command name that invokes the transformation, and a set of replacement actions. Formally, a transformation is written as:

transform selection_pattern *on*
command_name { replacement rules }

The selection pattern consists of zero or more hierarchical patterns that must match the objects that the user has selected on the display. In the event that there is more than one pattern, each pattern must match a different object that has been selected. A pattern is organized as a tree, with an object name or nonterminal at its root and the names of subcomponents (terminals and nonterminals) as children. Each of these subcomponents is in turn the root of its own tree. Figure 4 shows a sample pattern that would match a

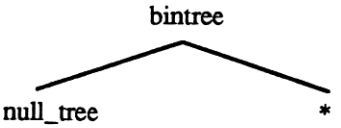


Figure 4: Sample selection pattern that matches a binary tree node that has a null left child and anything for the right child.

binary tree node whose left child is nonexistent. bintree and null_tree represent nonterminals in the constraint grammar, and * denotes the wildcard character that matches anything.

The selection pattern is compared against the selected portion of the encapsulated directed graph. If the pattern matches a portion of the subgraph that is rooted at the selected object, the transformation can be enabled by selecting the command name. For example, suppose the user selects the tree node shown in Figure 3.a. This node corresponds to the darkened node in the encapsulated directed graph (Figure 3.b). The pattern in Figure 3 does not match the subgraph rooted at this node. The root nodes match since both nodes are "bintrees". However, the left children do not match, since the pattern expects a "null_tree" but the child in the actual graph is a "bintree". If the user had selected the left child of this node instead, the match would have succeeded.

Selection patterns can be thought of as a set of conditions that the selected objects must satisfy if the transformation is to be enabled. For example, the binary tree application uses the selection pattern in Figure 3 to ensure that the user does not add a left child to a node that already has a left child. Those transformations whose selection patterns match the selected portions of the encapsulated directed graph are invoked by choosing the appropriate command name. Once invoked, the selected objects are modified by the replacement actions associated with the transformation. These commands typically perform operations such as creating or destroying objects, or replacing subcomponents.

For example, the CONSTRAINT system [14,15] provides three basic types of commands—*delete*, *create*, and *replace*. The *delete* command simply deletes an object if it has no parents. The object's subcomponents are recursively deleted if they have no other parents. The *create* and *replace* commands are directed by a hierarchical replacement pattern which is constructed in the same manner as the selection pattern. The *create* command builds the subgraph directed by the replacement pattern and inserts it in the appropriate place in the encapsulated directed graph. For example, the command

create(bintree(bintree(null_tree,null_tree),null_tree))

would create a subgraph that has the nonterminal bintree as its root node and the nonterminals bintree and null_tree as its successors. This subgraph corresponds to a binary tree with a left child.

In the *replace* command, the replacement pattern serves as the model subgraph that the selected portion of the

encapsulated directed graph should resemble when the replacement command terminates. The replacement pattern typically refers to portions of the selection pattern in directing these changes. This is done by refering to variables bound as a result of matching the selection pattern to the selected subgraph. For example, if the selection pattern is "bintree(left_child : bintree, right_child : bintree)", then the replacement pattern "bintree(right_child,left_child)" switches the order of the two subtrees of the selected tree.

CONCLUSIONS AND FUTURE WORK

Constraint grammars offer a promising model for UIMS's that seek to separate the graphical and nongraphical aspects of an application. By encapsulating the data structures needed by both the interface designer and application programmer, and by providing access to these data structures via transformations, constraint grammars allow the graphical and nongraphical aspects of the application to be designed and implemented independently. UIMS's that incorporate constraint grammars should be able to able to specify and automatically generate applications that manipulate complex data structures, such as program visualization systems, as well as applications currently implemented by UIMS's, such as simulation systems and programming environments.

In the future, the constraint grammar model will be enhanced by incorporating *inheritance hierarchies* and *structural constraints*. Structural constraints describe structural relationships between the components of an object; thus they can change internal nodes of the encapsulated directed graph as well as the leaf nodes and attributes. Structural constraints will be useful in automatically adjusting data structures such as balanced trees.

ACKNOWLEDGEMENTS

Much of the work in this paper was performed at Cornell University under National Science Foundation Grant DRC 86-02663. Production of the paper was supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

I wish to thank Brad Myers, Dexter Kozen, and Bill Pugh who were especially helpful in providing comments and insights during the development and presentation of the ideas in this paper.

REFERENCES

1. Barford, L.A. 1987.*A Graphical, Language-Based Editor for Generic Solid Models Represented by Constraints*. PhD thesis, Cornell University.
2. Barth, P.S. 1986. An object-oriented approach to grahical interfaces. *ACM Transactions on Graphics*, 5, 2 (Apr. 1986), 142-172.
3. Borning, A. 1981. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3, 4 (Oct. 1981), 353-387.
4. Hudson, S.E. 1986a. *A User Interface Management System Which Supports Direct Manipulation*. PhD thesis, University of Colorado, 1986.
5. Hudson, S.E. 1986b. Implementing a user interface as a system of attributes. In *2nd ACM SIGSOFT/SIGPLAN symposium on practical software development environments*, (1986), pages 143-149.
6. Knuth, D.E. 1968. Semantics of context-free languages. *Math. Syst. Theory*, 2, 2, (1968), 127-145.
7. Myers, B.A. 1987a. Creating dynamic interaction techniques by demonstration. In *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems*, 271-278.
8. Myers, B.A. 1987b. *Creating User Interfaces by Demonstrations*. PhD thesis, University of Toronto, Technical Report CSRI-196, May 1987.
9. Olsen, D.R. 1986a. Editing templates: a user interface generation tool. *IEEE Computer Graphics and Applications*, 6, 11 (Nov. 1986), 40-45.
10. Olsen, D.R. Jr. 1986b. MIKE: the menu interaction kontrol environment. *ACM Transactions on Graphics*, 5, 4, (Oct. 1986), 318-344.
11. Reps, T. and Teitelbaum, T. *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University, Ithaca, NY, 1988.
12. Stefik, M., Bobrow, D.G., and Kahn, K.M. 1986. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3, 1, (Jan. 1986), 10-18.
13. Sussman, G.J. and Steele, G.L., 1980. Jr. CONSTRAINTS—A language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14 (1980), 1-39.
14. Vander Zanden, B.T. 1988a. Constraint Grammars in user interface management systems. *Graphics Interface '88 Conference Proceedings*, (June 1988), Edmonton, Canada, June 6-10.
15. Vander Zanden, B.T. 1988b. Incremental Constraint Satisfaction and Its Application to Graphical Interfaces. PhD Dissertation, Cornell Univ., Ithaca, NY. 14853.