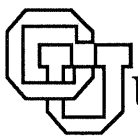
Constraints Provide Domain Behavior in a Construction Kit

Mark D. Gross and Casey Boyd

CU-CS-583-92 February 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Constraints Provide Domain Behavior in a Construction Kit

Mark D. Gross and Casey Boyd

CU-CS-583-92 February 1992

CONSTRAINTS PROVIDE DOMAIN BEHAVIOR IN A CONSTRUCTION KIT

Mark D Gross and Casey Boyd

Human-Computer Communication Group
Department of Computer Science
University of Colorado
Boulder, Colorado 80309
mdg@cs.colorado.edu; cboyd@cs.colorado.edu

ABSTRACT

We have added constraint management to a construction kit, making its direct manipulation interface faster, easier, and more natural-seeming to use. We began with the Janus construction kit, a design environment which has been oriented toward kitchen design by loading it with knowledge of the kitchen domain. Our constraint management code has allowed us to load the construction kit with knowledge of how we want objects in the layout to relate to one another and to behave. We report on how constraints produce partially self-assembling configurations, how this changes the construction kit interface, and the methods by which constraint knowledge is acquired.

KEYWORDS: constraints, direct manipulation, design process

INTRODUCTION

Constraints offer a mechanism to build user interfaces that embody and exhibit the behavior of a specific domain. In a construction kit environment where users assemble complex configurations from a simple palette of objects constraints can provide graphic objects with specific interactive edit behaviors. The environment can simulate the behavior in the layout of the objects. Our constraint system extends the Janus construction kit environment (reported on earlier [3, 4]) in several ways. First, construction objects have built-in behaviors that enable the designer to more quickly and easily operate in the construction environment. Second, our system provides the designer with constraint-defining tools to program the interactive edit behavior of design objects. Third, our system offers a "learn from examples" mode where the designer can show the system new constraint relations.

Since Sutherland's Sketchpad program [17], constraints have been used in various applications in computer graphics and design. Interactive constraint-based kits such as ThingLab explored the interaction of constraint-based and object-oriented programming and showed how constraints could be used to provide an interactive graphic environment with object behaviors [1]. Constraint based techniques have also been used in computer aided design [11,16,2,8]. More general work on constraint satisfaction (e.g. [7,12]) has supported the development of applications and constraint programming languages and environments [10,15].

For those who are unfamiliar with the Janus construction kit, a few words will provide some needed background. Colleagues in our laboratory have developed the Janus program, a prototype design environment to explore issues in computer supported cooperative design [5]. The two faces of Janus are (1) a construction kit in which a designer selects elements from a palette and combines them in a work-area to make configurations, and (2) a hypertext document of issues and arguments about the design domain, which offers guidance about selection and placement of elements.

The construction kit was developed around a specific example domain from architectural design: the layout of kitchens. The domain was chosen because it is small and a significant portion of knowledge can be coded. It is familiar to most people yet sufficiently rich to stimulate discussion. In kitchen layout and hence in Janus spatial constraints are most important. Although nonspatial constraints (e.g. cost, light, safety) play a role, ultimately the designer must select, size, and place the elements in the workspace. Thus many "higher level" constraints can be expressed and realized in terms of specific spatial constraints. For example, a safety constraint, "the kitchen should be safe from fire hazard," can be specified in terms of lower level spatial constraints: for example, "the stove should not be under the window" (because curtains might catch fire).

The next section describes the use of our system and explains the mechanisms that support it. Specifically, we discuss the use of inherent constraints to give objects default behavior, how the system resolves constraint conflicts that arise when objects interact, how users define new constraint types, how the system acquires constraints from examples, and the algebraic constraint representation that underlies the system's interactive behavior. We conclude with a summary and a discussion of further work.

CONSTRAINTS IN A CONSTRUCTION KIT

Constraints in the Janus construction kit environment enable the designer to assemble a design more quickly and easily. Without constraints, placement was a tedious task. With the constraints, objects first snap to a reasonable location according to their preprogrammed "inherent constraints" and the designer can then edit the layout to adjust the design. The following brief scenario illustrates the system's behavior.

1. In a typical design session the designer begins by placing four walls from the palette. The walls snap together at their corners to make a room. The four walls form a basis for the design and create a framework for further action. (figure 1a) The walls stay joined as the designer adjusts their dimensions.

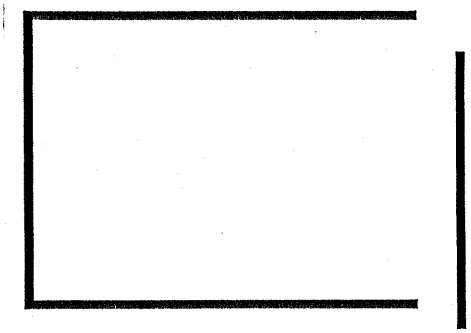


Figure 1a.

2. The next step brings in the doors and windows. These elements snap into walls as they are placed and can be dragged only along the lengths of the walls. (figure 1b) A constraint keeps each window and door in the plane of its wall but allows it to slide along the wall's length. By default the constraint between a wall and a window (or door) allows the wall to control the position of the window.

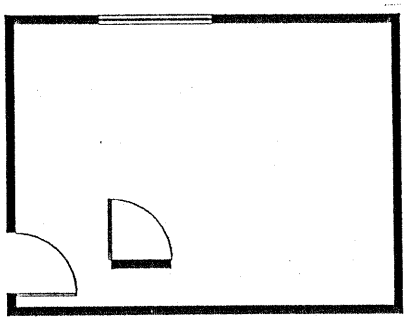


Figure 1b.

3. Next, the designer brings in major appliances: stove, sink, and refrigerator. These snap into position against the nearest wall along the shortest path from where the user released them, except the sink which snaps to center under the window. Also, the corner cabinets snap into place in the corners behind them. (figure 1c-d) Each of these elements has preprogrammed inherent constraints that govern their behavior. In each case, a default dependency determines which element in a relation will be dominant and which will be dependent.

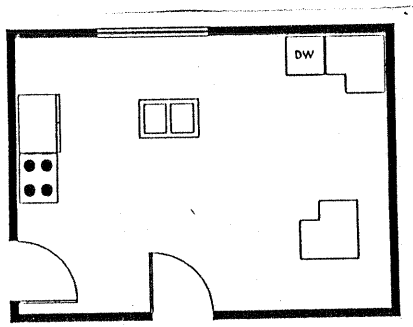


Figure 1c.

4. To complete the assembly, the designer brings in the remaining cabinets and counters, which snap to the wall, then slide along the wall to the nearest neighbor. (figure 1d)

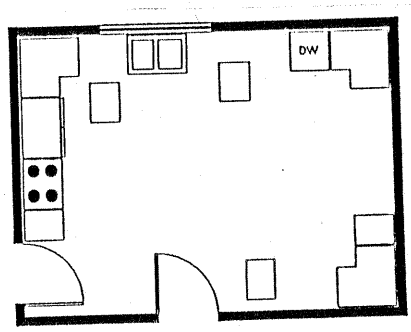


Figure 1d.

5. Finally, the designer edits the layout, adding constraints that allow the cabinets to stretch, and moving the appliances slightly to improve the design. (figure 1e)

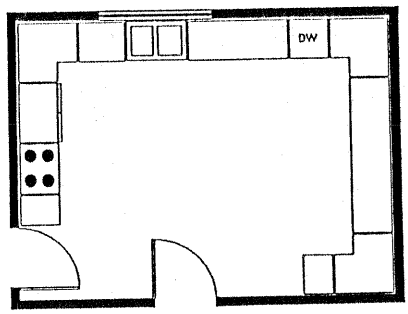


Figure 1e.

Inherent Constraints Provide Default Behavior

Constraints associated with each element class provide default behavior. We call these *inherent constraints* because they are applied automatically when a new element is instantiated. They describe the element's typical or expected behavior. In figure 1a, walls have inherent constraints that join them at their corners. In figure 1b, when the designer places a window near a wall, the window attaches itself in the wall. The window's inherent constraint places it in the wall but free to move along the wall's length. In figure 1c, the sink has an inherent "in front of window" constraint, and other appliances have an inherent "in front of wall" constraint. Of course, the designer can override the defaults or delete the inherent constraint.

An inherent constraint requires the system to find appropriate arguments in the work area for the new constraint. A sink placed in the work area must find a window to attach itself to. Specific procedural code belonging to each element class helps the system select arguments for inherent constraints and determine how constraints should be applied. For example, in figure 1a, procedural code associated with the "corners" constraint that joins two walls inspects the geometry of the walls in the work area and determines which two edges to relate. Four possible corner geometries lead to four possible sets of primitive algebraic relations. The procedure chooses the two closest edges and establishes algebraic constraints that bring and keep them together. The principle is that the user makes decisions and indicates them with rough positions and the system does the legwork.

Adding and Removing constraints

In addition to the automatically provided inherent constraints, it is easy to interactively apply constraints to the design and to remove them. The system provides a menu of commonly used spatial constraints that can be applied to any two elements. For example, an "adjacent" constraint applied to a sequence of elements keeps them aligned on one edge and adjacent. The designer can use this constraint to snap a selected set of elements together as a group and keep them lined up. Other constraints may be limited to certain types of elements. For example, a "sink-window" constraint cannot be applied to a refrigerator. Type-checking the arguments allows the system to limit the application of a constraint.

It is also easy to remove previously established constraints. After the user selects, drags, and releases an element, the constraint propagator engages. It either moves the element to a constrained position (perhaps returning it to its original position) or it moves another related element to maintain the established constraints. To move an element without the constrained behavior, the designer can select an element using a "remove-constraints" modifier key. Before dragging the element, all its constraints will be deleted from the network and the element may be freely placed.

Resolving Constraint Conflicts

Conflicts arise and must be resolved almost every time a new constraint is asserted [9]. Typically the new constraint asserts some condition that is not true in the current state of the design and the program must take action. For example, in figure 1c, when the designer brings in a sink from the palette its inherent constraint asserts that "sink in front of window." The new constraint conflicts with current positions of the sink and the window. To resolve the conflict the system must move the sink or the window and other objects related to them. A resolution mechanism determines how to restore the network to consistency.

A simple but effective resolution mechanism handles most conflicts in our system based on a partial ordering of element classes. Most constraints relate slot values of elements that belong to different classes, (e.g. a sink and a window). Domain knowledge provides a default resolution strategy: in architectural design some types of element are dominant to or "more fixed than" others. Windows are usually more fixed than sinks. Walls are more fixed than windows. By default, a window — a built-in element — will control the position of a sink — an appliance. If the window is moved the sink will follow, but if the sink is moved it will snap back to maintain its relation with the window. This is the default one-way dependency behavior referred to earlier. However the designer can override this one-way dependency, either for particular elements ("this sink should control this window") or reprogramming the default behavior of element classes.

Element classes are ordered with a "fixity" number assigned to each class. When constrained slot values in two elements conflict, the element with higher fixity wins. The conflict resolver retracts the other slot value and the solver then computes a new value consistent with the rest of the network. In the example above (figure 1c), the sink moves to take its place at the window. When a constraint relates two elements of equal fixity, they are mutually dependent: Whichever element is moved causes the other to accommodate.

This simple scheme handles many cases in kitchen layout. However, in some cases a spatial constraint should not be satisfied by moving an element, but by changing a dimension. A counter between a stove and a refrigerator might stretch as the designer separates the refrigerator and stove; refrigerators and stoves should not freely change their dimensions. We implement these preferences by attaching a fixity to each slot in addition to the element's overall fixity. A counter's size slot has a lower fixity than its edge position slots; when we constrain a counter adjacent to other elements it will stretch not move. The conflict resolution functions process these fixity numbers to determine which variable to relax.

Extending the Constraint Knowledge

Experienced designers use certain preferred relationships frequently. Designers can program these preferred relationships in two different ways, neither of which requires writing code. In the first method, the user defines a new constraint type by explicitly combining several existing ones, thereby determining for the system how the new constraint will be implemented. In the second method, the user lays out an example and the system determines how to implement the new constraint. Here as elsewhere rough positioning is sufficient.

Defining Constraint Types Explicitly

The designer can extend the built-in behaviors by defining new constraint types composed from existing ones. The designer defines a new constraint type by selecting already known constraints from menus and applying them to specific elements in the work area. The new definition collects the constraints and abstracts them from their

specific element arguments. The new constraint type is then added to a menu and is available for immediate application.

Acquiring Constraints from Examples

Constraints give the user a powerful way to program the construction kit with desired design behavior. However, it requires the user to explicitly specify the constraints. Specifying all the desired relationships can be tedious; therefore we have developed a way for our system to acquire knowledge of constraints based on observing relationships that the designer establishes [14,13]. This approach relieves the overhead often associated with programming of constraint-based environments.

The designer can program a new constraint by showing the system an example design configuration. In the course of a brief graphical dialogue with the designer the system tries to identify the precise relation that the designer intended, then it stores the induced relation for later use.

The designer selects two or more elements in the work area and gives the acquire constraint command. The system displays a graphical menu at the top of the work area in which each option represents a different interpretation of the selected configuration of elements. Each interpretation embodies a different set of spatial relations. For example, figure 2 shows three guesses as to the user's intended relationship between a sink and a window. The example is ambiguous about whether the sink is centered, left- or right-aligned in front of the window. The designer intended the sink to be centered in front of the window and will select the guess on the left. The system removes the menu and applies the new constraint relationship to the selected configuration. The sink centers itself beneath the window and moves adjacent to it. The system also adds the new constraint type to its menu of preprogrammed constraints. This can be done during a design in progress.

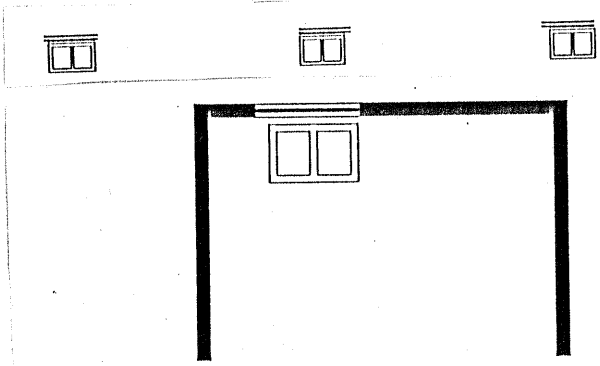


Figure 2: the system acquires new constraints from examples.

The algorithm for constraint acquisition is straightforward. It identifies a set of alternative parses of the relationships found in the configuration. The system takes the elements in the example configuration pairwise, in this case the sink and window. It compares each pair with a list of known spatial relations, and produces a sublist of relations that hold for the pair. In this example, three relationships are identified: left-aligned, right-aligned, and centered. When the example contains more than one pair of elements, the system produces a list of relations for each pair. The system then produces all permutations taking one relation from each pair. This constitutes the complete set of all candidate interpretations of the relations in the example.

The system rests on algebraic constraints

Each object has a set of slots that describe its attributes, including its x- and y- coordinates in the work area and its size. Table 1 shows the slots for a typical design element, stove-1.

```
#<FOUR-ELEMENT-STOVE FOUR-ELEMENT-STOVE-1 300002434>
 is an object of class
 #<CLOS:STANDARD-CLASS JC::FOUR-ELEMENT-STOVE 33151337>
  OC::NAME:                    FOUR-ELEMENT-STOVE-1
  JC::LEFT-X:                  78.0
  JC::TOP-Y:                   210.5
  JC::WIDTH:                   30
  JC::DEPTH:                   24
  JC::ROTATION:                0
  JC::IS-A                     JC::FOUR-ELEMENT-STOVE
  JC::IS-A                     JC::STOVE
  JC::IS-A                     JC::APPLIANCE
```

Table 1: slots in stove-1.

Algebraic constraints on object slot values enable a user to program the interactive edit behavior of design elements. Our constraint manager parses linear equations, constructs a network, and propagates values through the network. It provides functions for adding and deleting constraints to the network and for fixing and retracting variable values. Standard propagation techniques [6] are employed, with dependency maintenance to support retraction of constraints and values. We have not concentrated on building a powerful constraint solver; for example, it does not handle cycles in the constraint network. However the constraint mechanisms in our system are sufficient for the ideas explored in this paper.

Linear equations and slot values are appropriate for the constraint solver but not for the end user. The constraints described above are composed of these lower level algebraic primitives. Procedural code associated with each constraint type determines how to implement the constraint using algebraic primitives. That code is run each time a constraint is applied. For example, when a stove is attached to a wall, the system asserts the equation:

```
stove.top-y = horizontal-wall.top-y +
horizontal-wall.depth + 1
```

With this constraint the stove moves to a position adjacent to the wall. In this case adjacent means placing the top edge of the stove one unit below the bottom of the wall (i.e., the top edge of the wall plus the wall's depth). When two elements are constrained in this fashion, they maintain this relationship thereafter. If the wall is moved (to enlarge or reduce the kitchen's size), the stove will move along with it. A variable changes and the system adjusts other variables to restore network consistency.

DISCUSSION AND FURTHER WORK

What role should constraints play in a design environment and how can they be effectively integrated into a construction-kit? In our system constraints animate the design. Constraints make it easier and faster to assemble a layout, because elements tend to find their place in the design automatically. It streamlines the process, eliminating what had been awkward and tedious in Janus. A key issue is whether users consider the elements to be well-behaved, and whether it is easy for users to change an element's behavior if they do not like it.

Embedding constraints into the construction environment simplifies the layout task but the need to program the behavior adds a potential for complication. We would like designers to use constraints to program design behavior without learning a complicated constraint programming language. It is important that a designer be able to easily define new constraint types to extend the built-in constraints and to specify mechanisms to resolve constraint conflicts. The program must acquire knowledge — in the form of constraints — about the domain in general and about the developing design in particular. The system can also acquire new constraint types by examining examples provided by the designer and it can also be programmed explicitly by combining existing constraints.

The domain of kitchen design has served us well as an initial testing ground for our ideas. The small and finite set of elements and relations made it a comprehensible domain. How will our ideas extend to domains where the problems are larger and more complicated?

We have chosen as a successor domain the problem of Local Area Network design and layout in buildings. We are interested in this domain because it has an architectural component yet it requires new and fundamentally different behaviors which we think can be modeled by constraints. In addition to position and size of architectural elements, constraints will control the selection and connection of network hardware. Workstations, transceivers, thinnet and thicknet cables must be assembled giving consideration to cable length limits, network topologies, and connectivity requirements. As the network design develops and evolves, the constraint management system will continue to support the designer by maintaining consistency and correctness. The value should be greater as the interactions between elements are more intricate and less visible.

SUMMARY

We are using our system to explore the effective integration of constraint representations for design knowledge and behavior into construction kit environments. Our constraint system adds to the Janus construction environment a way to program design elements with inherent interactive behavior. Appliances stick to walls, sinks find their place under windows, counters stretch, and elements line up. The system's inherent constraints enable a designer to program the design elements with placement and layout knowledge. Designers can extend our system by interactively defining their own constraints.

A simple algebraic constraint management system underlies our extensions to the design environment. We described a facility for defining constraints, a conflict resolution procedure that uses an ordering of object classes to select which variable to relax, and our scheme of packaging inherent constraints in object classes that are applied when new elements are instantiated. To lighten the burden of explicitly describing all the relations to be maintained in a design, our system acquires new constraint relations by observing configurations that the designer makes. The result is an interface that is smoother, faster, less tedious, more natural-seeming, and easier to use.

We are currently working on a LAN layout editor based on the ideas we have developed. We expect that our approach will extend naturally to the new domain but will offer new challenges for constraint solving and conflict resolution.

REFERENCES

1. Borning, A. Programming Language Aspects of ThingLab. ACM Transactions on Programming Languages and Systems, 3,4, (1981) pp. 353-387.

2. Bowen, J., and O'Grady P. Constraint Networks for Life-Cycle Engineering. In: Proc. AI & Engineering '90. (Boston, MA, 1990), Computational Mechanics Press, pp. 281-296.

3. Fischer, G., and Lemke, A. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. Human-Computer Interaction, 3,3 (1988) pp. 197-222.

4. Fischer, G., and Morch, A. JANUS - Integrating Hypertext with a Knowledge-based Design Environment. In: Proc. Hypertext '89 Proceedings. (1989), pp. 105-117.

5. Fischer G., and Nakakoji, K. Empowering Designers with Integrated Design Environments. In: Artificial Intelligence in Design. J. Gero, Ed. Computational Mechanics Press, 1991.

6. Freeman-Benson, B., Maloney, J., and Borning, A. An Incremental Constraint Solver. CACM, 33,1 (1990) pp. 54-63.

7. Freuder, E. Synthesizing Constraint Expressions. Communications of the ACM, 21,11, (1978) pp. 958-966.

8. Gross, M. Knowledge-Based Support for Subsystem Layout in Architectural Design. In: Artificial Intelligence in Engineering: Design. Gero J, Ed. Computational Mechanics Press, Southampton, UK, 1990.

9. Klein, M., and Lu, S. Conflict resolution in cooperative design. Artificial Intelligence in Engineering, 4,4 (1989) pp. 168-180.

10. Leler, W. *Constraint Programming Languages.* Addison Wesley, Boston, 1987.

11. Lin, V.C., Gossard, D.C., and Light, R.A. Variational Geometry in Computer Aided Design. Computer Graphics (SIGGRAPH -81 Proceedings), 15,3 (1981) pp. 171-177.

12. Mackworth, A. Constraint Satisfaction. In: Encyclopedia of Artificial Intelligence. S. Shapiro, Ed. Wiley, NY, 1987, pp. 205-211.

13. Maulsby, D., and Witten, I. Inducing Programs in a Direct-manipulation Environment. In: Proc. Human Factors in Computing (SIGCHI '89). (1989), ACM Press / Addison Wesley, pp. 57-62.

14. Myers, B. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

15. Myers, B. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. In: Proc. Human Factors in Computing Systems (SIGCHI '91). (New Orleans, 1991), ACM Press / Addison Wesley, pp. 243-249.

16. Sapossnek, M. Research on Constraint-Based Design Systems. In: Proc. 4th Intl. Conf. Applications of AI in Engineering. (Cambridge, England, 1989).

17. Sutherland, I. Sketchpad - a Graphical Man-Machine Interface [Ph.D. Dissertation]. M.I.T., 1963.