The Importance of Pointer Variables in Constraint Models

Brad Vander Zanden

Computer Science Department
University of Tennessee
Knoxville, TN 37996
bvz@cs.utk.edu

Brad A. Myers
Dario Giuse

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
brad.myers@cs.cmu.edu
dzg@cs.cmu.edu

Pedro Szekely

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
szekely@isi.edu

Abstract

Graphical tools are increasingly using constraints to specify the graphical layout and behavior of many parts of an application. However, conventional constraints directly encode the objects they reference, and thus cannot provide support for the dynamic runtime creation and manipulation of application objects. This paper discusses an extension to current constraint models that allows constraints to indirectly reference objects through pointer variables. Pointer variables permit programmers to create the constraint equivalent of procedures in traditional programming languages. This procedural abstraction allows constraints to model a wide array of dynamic application behavior, simplifies the implementation of structured object and demonstrational systems, and improves the storage and efficiency of highly interactive, graphical applications. It also promotes a simpler, more effective style of programming than conventional constraints. Constraints that use pointer variables are powerful enough to allow a comprehensive user interface toolkit to be built for the first time on top of a constraint system.

Keywords: Constraints, development tools, incremental algorithms

1 Introduction

User interface toolkits, particularly graphical layout tools, are increasingly adopting the constraint model of computation. The constraint model uses equations to denote relationships between two or more objects. For example, a designer might write the following equation to position a circle 10 pixels to the right of a rectangle:

```
left = my-rect.right + 10
```

The advantage of the constraint model is that changed data is automatically propagated to the appropriate places and relationships are automatically maintained. Thus, if a user drags the rectangle around the screen, a constraint solver continuously resatisfies the above equation, causing the circle to follow the rectangle around the display.

Having a constraint solver automatically maintain a set of relationships is obviously advantageous, since it saves the programmer from having to manually write code to accomplish the same task. However, conventional constraints directly encode the objects they reference. For example, `my-rect` is hardcoded into the above constraint. Thus they cannot support the dynamic runtime creation of objects, since the constraints in the newly created objects will reference the wrong objects. These constraints also cannot model any application behavior which involves objects that continuously change their relationships to other objects. For example, a feedback object must highlight any item in a menu, an arrow might point at different boxes in a boxes-and-arrows editor, and a truck switches streets as it navigates through a city.

These shortcomings can be remedied by adding pointer variables to constraints that allow objects to indirectly reference other objects (these constraints are called *indirect reference constraints*). For example, the above constraint could be rewritten as[1]:

```
left = self.obj-over.right + 10
obj-over = my-rect
```

where `self` refers to the object containing the variable `left`, in this case, the circle. By changing the value of the

[1]To improve readability, we are expressing constraints in conventional infix notation rather than Lisp's prefix notation. In Garnet the constraint would actually be written as `(+ (gvl :obj-over :right) 10)` where `gvl` stands for *g*et *v*alue through *l*ink. The ':' that goes before variable names is also dropped in the notation used in this paper.

variable `obj-over`, the circle can be positioned to the right of any object. With this extension, constraints can support the runtime creation of objects and express the dynamic behaviors that occur inside an application window, as well as the static layout relationships that occur around the application window.

Pointer variables allow the programmer to define the constraint equivalent of procedures in traditional programming languages. Generalizing direct references into indirect references is akin to defining the parameters of a procedure. The advantages of procedural abstraction are well known, but the implementation of procedural abstraction in constraint systems is still quite novel. Constraint procedural abstractions greatly simplify the implementation of many interface features and enable the implementation of new ones that would have been unwieldy without procedural abstraction:

- Feedback, in which objects, such as checkmarks or inverted rectangles, may appear with any item in a set of objects;
- Prototype-Instance models, in which instances of constraints must be inherited from prototypes and references must be adjusted so that they point to the instance rather than the prototype;
- Programming by example, in which constraints that are demonstrated for example objects must be converted to general constraints that work with any object;
- Abstract specification of layouts, in which generic objects are laid out using constraints, and the specific widgets are filled in later, based on such parameters as the availabilty of screen space;
- Simulations, in which objects are frequently constrained to new and different objects, for example objects moving between the machines on a factory floor.

Over the last couple years, we have gained considerable experience using indirect reference constraints in the Garnet project [11]. We have found that they are crucial for implementing the insides of application windows, which is the hardest and most time-consuming portion of an interface to construct. In Garnet, constraints consist of arbitrary pieces of Lisp code and consequently, they are used to specify more than just graphical layouts. For example, they are used to communicate information between multiple threads of a dialog, to compute the attribute values of objects, and to monitor the states of various objects. The procedural abstraction provided by indirect reference constraints is so powerful that Garnet implements its toolkit on top of the constraints [11]. No other constraint-based toolkit does this.

Indirect reference constraints also provide an entirely new style of programming that seems much simpler and more effective than conventional constraints. It will become apparent how indirect reference constraints lead to far simpler implementations as this paper describes many of the important applications of indirect reference constraints. This paper will also discuss how indirect reference constraints can enhance the performance of an application while decreasing its storage demands. Finally, various implementation strategies for indirect reference constraints will be discussed.

2 Related Work

While pointer variables are commonly incorporated in programming languages, they have only recently been incorporated in their full generality in constraint systems. A restricted version of indirect reference constraints first appeared in Coral [17]. Coral permitted a designer to provide a list of objects that a constraint could reference. For example, a designer could provide a list of menu items and a feedback object would be able to appear over any of them. However, Coral did not allow constraints to reference arbitrary objects through variables, and thus did not provide the full generality of indirect reference constraints.

Thinglab [2] also provides a limited form of indirect reference constraints. Designers can construct pathnames that allow a constraint to traverse a structure hierarchy to find an object. If one of the components in the structure hierarchy changes, the new object will be automatically referenced by the constraint. However, arbitrary references to objects through pointer variables are not supported. Penguims [7] supports a model of indirect reference constraints that is similar to the one described in this paper but it uses a different constraint solving algorithm. Many other systems, such as Grow [1], Apogee [5], Peridot [9] and CONSTRAINT [19], allow constraints to directly reference objects but do not allow indirect references.

Kaleidoscope supports a different type of abstraction—constraint abstraction rather than procedural abstraction—in which procedures consist of a set of parameterized constraint statements and produce as output a set of constraints instantiated with the parameters passed to the procedure [3].

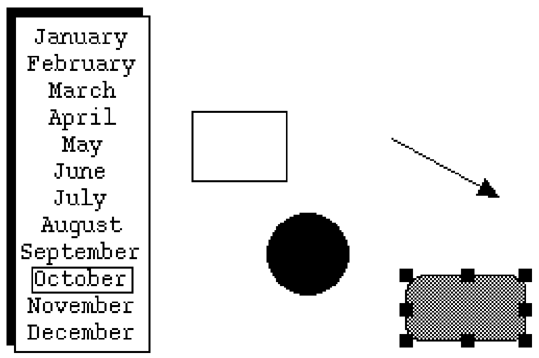
Finally, a number of researchers have developed models that allow constraints to have variables, but not pointer variables [16, 15, 8, 14]. For example, a programmer could write a constraint such as *feedback.position = item1.position + offset*, initially assign the value of 10 to `offset`, and later assign the value of 20 to `offset`. However, a programmer could not write a constraint of the form *feedback.position = self.obj-over.position + offset*, where `obj-over` is a pointer variable that points to an arbitrary object.

3 Applications of Indirect Reference Constraints

Indirect reference constraints can be used to implement many parts of an application that are difficult or infeasible to implement with direct reference constraints. These include feedback, copying and instancing of composite objects with constraints in them, programming by example, abstract specification of layouts, and simulations.

3.1 Feedback

Most direct manipulation interfaces provide feedback to the user while performing an operation. For example, a rectangle may highlight the item that the user is currently pointing at in a menu (Figure 1.a). While it is generally impractical to handle feedback objects using direct reference constraints, they are easily handled using indirect reference constraints. For example, the feedback object in Figure 1.a must be able to highlight any of the menu items, but a direct reference constraint will only allow it to highlight one of these items. In contrast, indirect reference constraints allow the feedback object to reference any of these menu items through a variable, such as `obj-over`. This technique works equally well for feedback objects that highlight a fixed set of objects, such as the objects in a menu, or a dynamic set of objects, such as the objects in a drawing window (Figures 1.a and 1.b).



(a) (b)

Figure 1.

The rectangular feedback object in the menu and the selection handles in the drawing editor use constraints to center themselves over the selected items and to change their dimensions to the dimensions of the selected item. By indirectly accessing a selected item through the variable `obj-over`, the feedback objects are able to appear both over any item in a static set of objects, such as the menu items (a), or any item in a dynamic set of objects, such as the objects in the drawing editor (b).

3.2 Structured Objects

Pointer variables simplify the integration of constraints into a structured object system. A structured object consists of several parts, such as the labeled box in Figure 2, which consists of a rectangle and a piece of text. Typically these parts are mutually constrained. For example, the label is centered inside the box and the size of the box depends on the size of the label.

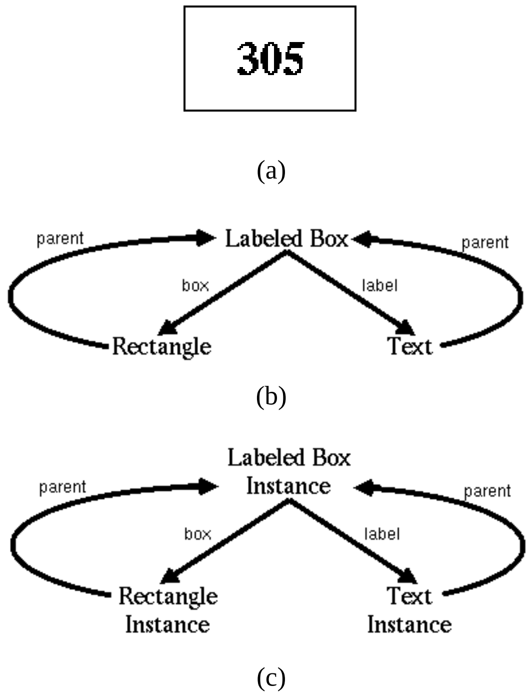


Figure 2.

Structured objects, such as this labeled box (a), are built up from other objects, such as this rectangle and number (b). Each object maintains pointers to its parent and its children, so that constraints can indirectly reference one another through pointers. This facilitates the copying and instancing of objects, since the object system simply sets the pointers in the new objects, and the constraints automatically reference the appropriate objects (c).

Interactive applications need to make copies or instances of these objects at runtime (e.g., creating new objects in a drawing program, creating new circuit elements in a circuit simulation program). These operations can be easily implemented using indirect reference constraints, but are quite difficult to implement in regular constraint systems.

In an indirect reference constraint system, each object maintains a pointer to its parent, and a set of pointers to its children (Figure 2.b). Constraints reference objects by following the appropriate pointers. For example, if the label's parent pointer is contained in the variable `parent` and the labeled box keeps pointers to its children in the variables `label` and `box`, then the label can be centered inside the box using the following constraints:

```
center-x = self.parent.box.center-x
center-y = self.parent.box.center-y
```

To create an instance of an object, the object system creates instances of each of the object's components and sets the pointer variables (Figure 2.c). The object system also creates instances of each of the constraints in the prototype's components and stores them in the appropriate places in the new instance's components. No changes are needed to the constraint expression. The constraints in the newly created objects will automatically reference the appropriate objects since they will follow the pointers in the instance objects rather than in the prototype objects. For example, the constraint that computes the value for `center-x` in the label instance will follow the `parent` and `box` pointers in the labeled box structure hierarchy and retrieve the `center-x` value of the rectangle instance.

In a direct reference system, constraints must use hardcoded references to objects. For example, the label in Figure 2 could be centered inside the box using the following direct reference constraints:

```
center-x = box.center-x
center-y = box.center-y
```

When a new instance of labeled-box is created, the object system will have to replace all references to box with references to the newly created instance of box.

The object system will have to track down the references by manually traversing the prototype's hierarchy to find where box is in relation to label, (the relation is go to label's parent, which is labeled box, then to labeled box's first child, which is box), then use the same traversal in the instance's hierarchy to find the appropriate reference to the newly created instance of box. Thus it is much simpler and more efficient to implement copying and instancing operations in indirect reference systems than in direct reference systems.

3.3 Programming by Example

Indirect reference constraints make it easier to implement systems that employ demonstrational programming, such as the graphical interactive design tool Lapidary [10]. In a demonstrational system, a user draws an example picture or demonstrates an example behavior, and then the system creates a prototype object or behavior by generalizing the picture or demonstrated behavior. If the demonstrational system uses indirect reference constraints, then it is easy to generalize these examples. In fact, the example that the user draws or demonstrates is already a prototype, since the object can be instanced or copied using the scheme described in the previous section.

In Figure 3, a designer is using Lapidary to create a boxes-and-arrows editor. The designer has drawn an example picture in which the arrows are attached to the center of the boxes they connect. Lapidary represents the constraints of the line internally as indirect reference constraints:

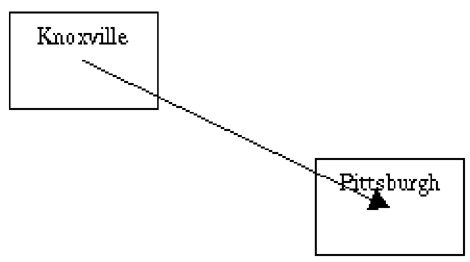


Figure 3.

An example picture demonstrating that the endpoints of an arrow should be attached to the centers of the boxes it connects. An interface builder will generalize this arrow into a prototype that can connect any pair of boxes.

```
endpt1 = self.from-obj.center
endpt2 = self.to-obj.center
from-obj = box1
to-obj = box2
```

The designer can save this arrow and the application can use it as a prototype. When the boxes-and-arrows editor creates instances of this arrow, it stores pointers to the appropriate boxes in the `from-obj` and `to-obj` variables, and the constraints automatically attach the endpoints of the arrow instance to the centers of the boxes. The application does not need to know anything about the constraints, structure, or graphics of the line. The constraints on the endpoints could connect centers to centers, right sides to left sides, or even use a complex formula that computes the nearest sides and tries to avoid crossing other lines.

3.4 Abstract Specification of Layouts

Indirect reference constraints facilitate the specification of layouts, independently of the objects to be layed out. For example, a designer might want to specify that a generic feedback object should appear over a selected object. The actual type of feedback used might depend on the size of the selected object and the type of the selected object. As another example, Humanoid [18] and Jade [20] allow a designer to define constraint-based rules that describe the general layout of dialog boxes in terms of the generic parts of a dialog box, such as a title, a body that contains the items of the dialog box, an OK button, and a cancel button. One can then apply the rules to different dialog boxes, irrespective of the widgets that fill the roles of the different parts. For example, storing the following constraint on the `x` coordinate of the OK button will force the button to be placed 10 pixels to the right of the dialog box's title, regardless of which widget is used for the title or the OK button:

```
left = self.parent.title.right + 10
```

3.5 Simulations

Simulations often require objects to move smoothly between various points of the display. For example, sort animations show objects moving around in linked lists or arrays, navigation systems move objects around transportation corridors, and manufacturing systems route objects through the machines on a factory floor. Indirect reference constraints model this motion by using variables to reference the beginning and target positions.

For example, suppose we want the carton in Figure 4 to glide from station A to station B as if it were on a conveyor belt. This could be done by writing a set of constraints that interpolate the carton's position based on a timer and the stations' positions:

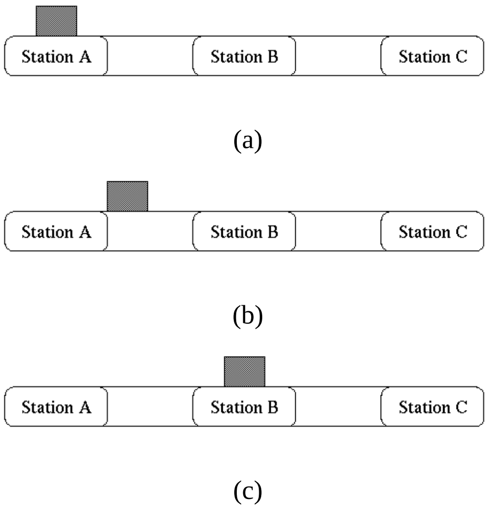


Figure 4.

An assembly line with stations connected by a conveyor belt. A carton should be centered above the station that is currently processing it (a), and cartons should move smoothly from one station to the next, (b) and (c).

```
time = 0
x-distance = self.to-station.center-x
             - self.from-station.center-x
center-x = self.from-station.center-x
         + (self.x-distance * self.time)
to-station = station-b
from-station = station-a
```

`x-distance` computes the distance between the old and new stations, and `center-x` contains the current x position of the carton. As the application program increments `time` from 0 to 1, the carton moves smoothly between its old and new assembly station. To prepare for the next step of the animation, the application resets the timer to 0, stores station B in `from-station`, and stores a new station, C, in `to-station`.

4 Performance and Implementation Advantages of Indirect Reference Constraints

The generalization of constraints using pointer variables can improve the efficiency of an application by reducing the number of constraints and objects it uses, reducing the size of the constraints it uses, and reducing the number of constraints that it must dynamically create and delete. Indirect reference constraints also make it easier for the constraint system to maintain one rather than multiple copies of a constraint and make it easier to statically compile the constraints.

Storage improvements come in two forms. First, by allowing objects to be constrained to many different objects, indirect reference constraints may significantly decrease the number of objects which an application creates. For example, suppose a feedback object should highlight the currently selected item in a menu, as in Figure 1.a. If direct reference constraints are the only constraints available, the designer may create a separate feedback object for *each* menu item, since the constraints will bind each feedback object to exactly one item. However, as noted in Section 3.1, indirect reference constraints allow a feedback object to highlight any menu item, and thus one feedback object suffices. Second, indirect reference constraints can be written much more compactly and elegantly than direct reference constraints. Returning to the feedback example, a clever designer who is working with direct reference constraints might be able to use only one feedback object by defining constraints which reference every object in a menu and describe how the feedback object should highlight each menu item. For example, to implement the feedback object in Figure 1.a, the designer might write the following constraint to define the left side of the feedback object:

```
feedback.left = case month
                 "January": Jan.left
                 "February": Feb.left
                        ...
                 "December": Dec.left
```

where month is a string variable containing the currently selected month.

However, this solution has four drawbacks:

- Non-modular and Inelegant: If the designer adds a new menu item, the designer must also remember to modify the constraints in the feedback object.

- Space: The constraint must have twelve separate conditions and actions, which causes the code to occupy a considerable amount of space at runtime. Also, 12 dependency pointers, one for each object, must be maintained by the constraint system.
- Efficiency: The constraint depends on all twelve objects. If one object changes, even if it is not the currently selected object, the constraint must be reevaluated.
- Dynamic Sets: This technique only works for static sets of objects, since the objects must be hardcoded in the constraint. It cannot be used to describe dynamic sets of objects, such as the objects in the drawing window in Figure 1.b.

Indirect reference constraints suffer from none of these disadvantages. The corresponding indirect reference constraint would be:

```
feedback.left = self.obj-over.left
```

where `obj-over` is a pointer to the selected menu item.

This constraint is both compact and modular. The designer can add or delete items from the menu without worrying about the effect of the change on the feedback object. It occupies less space than the corresponding direct reference constraint and requires only two dependency pointers, one to the variable `obj-over` and one to the selected menu item. It depends only on the `obj-over` variable and the currently selected menu item, so it will only be reevaluated when absolutely necessary. Finally, this type of constraint can handle dynamic sets of objects. If additional items are added to the menu, the constraint automatically deals with them without having to be rewritten.

The efficiency advantage of indirect reference constraints derives in part from their storage advantage. Fewer constraints means fewer constraints to solve, and thus, less work for an equation solver. For example, a constraint solver that employs eager evaluation might take only one fifth the time to set up the feedback for a five item menu using indirect reference constraints instead of direct reference constraints, because there are only one fifth as many constraints to solve.

Efficiency advantages also arise because 1) an object system does not have to locate and replace direct references; and 2) fewer constraints have to be dynamically created and destroyed. The search and replace issue was discussed in section 3.2. To illustrate the reduction in dynamically created and destroyed constraints, consider the construction of a menu using direct reference constraints. It may be impractical from a storage standpoint to maintain one feedback object for each item. Thus, the application may maintain only one feedback object and destroy the old constraints and create new constraints each time the feedback moves to a new item. The overhead involved in destroying and creating these constraints can be avoided if indirect reference constraints are used.

Indirect reference constraints also simplify the construction of the constraint system. First, the formula for an indirect reference constraint can be stored in a prototype and instances of the prototype can maintain pointers to this formula. Thus, many instances of a prototype constraint can be created, but the formula is created only once. Second, the parameters to a constraint are implicitly declared by pointer variables, so the constraint system can statically compile constraints by wrapping a function header around them. This considerably simplifies implementing a constraint system in an existing general-purpose language. For example, Garnet constraints can be arbitrary Lisp code. Direct reference systems typically also maintain only one copy of a constraint and statically compile it. However, to accomplish this, they require the user to write the constraint as a function, complete with parameters denoting the direct references, or else parse the constraint to determine the direct references. However, we have discovered that users find it irritating and cumbersome to have to define parameters for constraints. For example, it is much more elegant and compact to write

```
feedback.left = self.obj-over.left
```

than to write

```
constraint left-align (obj) { obj.left }
feedback.left = left-align(self.obj-over)
```

Similarly, it can be quite difficult to write a parser to search through each constraint and locate the direct references.

5 Implementation

The algorithms for implementing indirect reference constraints build on the algorithms for implementing direct reference constraints.

5.1 Lazy Evaluation

A variation of nullification/reevaluation algorithms can be used to handle indirect reference constraints. Nullification/reevaluation algorithms represent the constraints as a directed graph with nodes representing constraints, and edges (called dependencies) representing data flowing from one constraint to another (Figure 5.a). When the value of a node changes (typically a ‘‘leaf’’ node such as *e* or *f*), all nodes that directly or indirectly depend on this changed node are marked out of date. When the value of a node is requested, the constraint that computes its value starts demanding the values of other nodes on which it depends. If these nodes are out of date, they will recursively demand the values of the nodes they depend on, until eventually nodes are reached whose values are up to date, at which point the constraints can compute their value and return [13, 6]. For example, suppose that node *e* is changed in Figure 5. The lazy evaluator will mark the nodes *b*, *d*, and *a* as out of date (Figure 5.b). If the value of node *a* is then requested, *a* will demand the value of *d*. *d* is out of date so it demands the values of *e* and *f*, both of which are up to date, computes its own value, marks itself up to date, and returns its value to *a*. *a* then demands the value of *c* which is up to date, computes its own value, marks itself up to date, and returns.

Nullification/reevaluation algorithms were originally constructed with the assumption that the edges in the graph remain static while the constraint solver is evaluating the graph. However indirect reference constraints can cause the graph to dynamically change as the constraints are being evaluated, because the pointer variables may change, causing a constraint to access information from a different set of nodes. For example, when the constraint on node *a* is being evaluated, it may start referencing node *b* rather than node *c* (Figure 5.c).

To handle this situation, we have extended the algorithm so that dependencies can be dynamically created and deleted as the constraints are being evaluated. Dependencies are timestamped so that if they are not used by a constraint in a subsequent evaluation, they become stale and are discarded. When a constraint demands a variable, the constraint solver either creates a new dependency between the constraint and the variable if such a dependency did not already exist, or else updates the time stamp on the dependency so that it matches the timestamp on the constraint (a constraint is timestamped each time it is evaluated). The constraint solver removes stale dependencies as it invalidates constraints. Before following a dependency, it checks whether the dependency's timestamp matches the timestamp of the constraint it points to. If the two timestamps disagree, the dependency is discarded. A beneficial side effect of this scheme is that constraints which involve conditionals depend only on the variables that make up the condition and the branch of the condition that is executed. Thus the number of dependency pointers and unnecessary evaluations are minimized.

To see that this scheme works, note that a constraint will dynamically add or delete dependencies only if it contains pointers or conditionals. If a constraint depends on pointer variables, the constraint will be marked out of date when the pointer variables change and the constraint will be reevaluated when its value is next requested. At this point, the constraint solver will add edges to this constraint from the new set of nodes it references (Figure 5.c). The dependencies to variables that are not requested by the constraint on this evaluation will become stale and be removed the next time these dependencies are examined. Thus the constraint will demand the values of the correct set of nodes and will obtain the correct result.

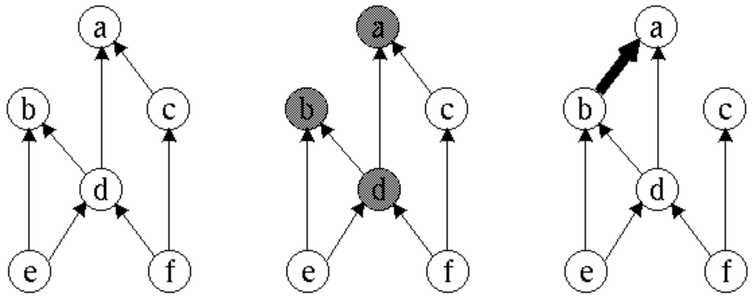
In the case of a conditional, the branch or branches of the conditional that were ignored during the previous evaluation of the constraint will only have to be evaluated if the condition itself changes. Since the constraint depends on the variables in this condition, it will be marked out of date when one of these variables changes and will be automatically reevaluated (of course it will also be reevaluated if one of the variables in the branch that was last executed is changed). Again, the constraint solver will add dependency edges to this constraint from the new set of variables it references in whichever branch is executed and remove edges that emanante from variables in the previously executed branch. Thus constraints with conditionals will always be evaluated correctly.

5.2 Eager Evaluation

Our eager evaluation algorithm uses a variation of an eager evaluator developed by Roger Hoover [4]. Like the lazy evaluator, this algorithm makes use of dataflow graphs. However, it assigns priorities to the nodes in the graph, indicating the nodes relative position in topological order (Figure 6.a). When a node changes value, all its immediate successors are added to a priority queue based on their priorities. When the evaluator starts executing, it removes the lowest priority node from the queue and evaluates it. By evaluating the lowest priority node in the queue, the evaluator ensures that the values of all nodes that the constraint associated with this node may request are up to date.

In the Hoover algorithm, the priorities are maintained in an ordered list and each node in the dataflow graph points to one of these priorities. Comparisons between priorities can be performed in O(1) time and insertions of new priorities can be accomplished in amortized O(1) time. When an edge is added to a dataflow graph, the algorithm checks whether the priority of the source node is greater than or equal to the priority of the destination node (data flows from the source node to the destination node; for example *e* is the source node and *h* is the destination node for the edge that connects these nodes). If the priorities are out of order, the algorithm follows the successors of the destination node transitively until it reaches nodes whose priority numbers are greater than the priority of the source node (the Hoover algorithm will also follow predecessors of the source node; however, to save space we do not maintain backpointers and thus we cannot search backward from nodes). These nodes are termed boundary nodes. The algorithm works back from these boundary nodes and assigns to the intermediate nodes new priority numbers that are between the priority numbers of the source and boundary nodes. If it runs out of existing priority numbers, it creates new ones by inserting records into the ordered list directly after the record associated with the priority number of the source node.

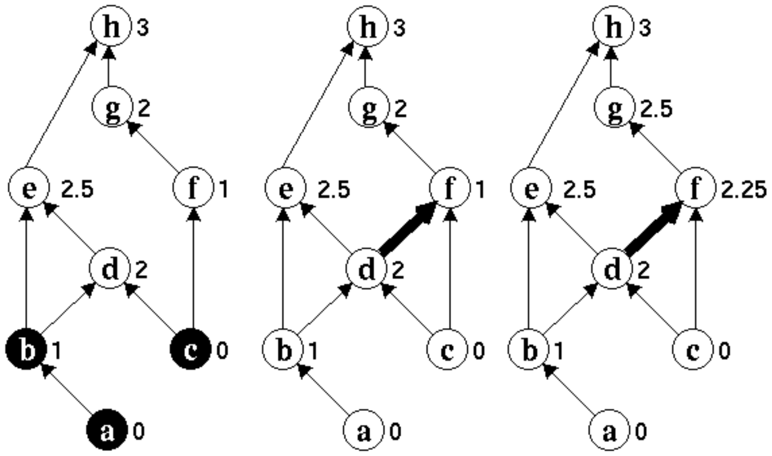
For example, suppose a dependency from node *d* to *f* is added in Figure 6.b. Node *d* has priority 2 while node *f* has priority 1 so the nodes are out of order. The algorithm goes to node *g*, which has a priority of 2, and then to node *h*, which has a priority of 3. Since this is greater than node *d*'s



(a) (b) (c)

Figure 5.

(a) Constraints are represented as nodes in a directed graph. The edges represent data computed by a constraint that another constraint uses. (b) The gray nodes represent nodes marked out of date when node *e* is changed. (c) Node *a* now depends on node *b* rather than node *c*.



(a) (b) (c)

Figure 6.

(a) Numbers are assigned to nodes according to the order in which they are evaluated. Nodes cannot be evaluated until all their predecessors have been evaluated. Darkened nodes represent evaluated nodes. Nodes *d* and *f* are ready for evaluation. (b) Node *f* now depends on node *d* as well as node *c*. (c) Nodes *f* and *g* must be renumbered to make their priorities agree with their position in topological order.

priority, the search stops here and node *h* becomes a boundary node. In this case there is one priority, 2.5, between 2 and 3 in the priority list, so the algorithm assigns this priority to node *g*.[2] At this point the algorithm runs out of existing priorities, so it inserts a new one in the ordered list and assigns it to node *f* (Figure 6.c).

The Hoover algorithm assumes that dataflow graphs cannot change once constraint evaluation begins, so the reordering scheme and the evaluator can be invoked in sequence. However indirect reference constraints may cause the edges of the graph to change *during* constraint evaluation. Thus

[2]We are using rational numbers for illustrative purposes only. The actual algorithm for maintaining the ordered list uses only integers and does some reordering to ensure that only integers are used.

the numbers assigned to the nodes may become incorrect and force an equation to be evaluated prematurely. To overcome this difficulty, we have taken the approach of dynamically updating the topological order each time the graph changes, and evaluating nodes according to this revised topological order. In other words, the reordering algorithm and the constraint evaluator are interleaved. Dependencies are dynamically created and deleted in the same way as in the lazy evaluation algorithm.

The time complexity of this algorithm depends on the number of reorderings and the time required by each reordering. Assuming a bounded number of variables per equation, a reasonable assumption for constraints in graphical interfaces, a reordering requires O(k) time where *k* is the number of nodes that must be reordered (Hoover's algorithm has an additional log factor in the running time since it uses a priority queue to guide its forward and backward searching; however this forward and backward searching may reorder fewer nodes than our algorithm, thus offsetting the log factor). Assuming there are *n* nodes in the graph, the worst case running time of the algorithm is O(dn) where *d* is the number of dynamically added edges. As with most incremental algorithms, this worst case running time is misleading, since most node evaluations do not trigger a reordering and most reorderings do not visit all *n* nodes. Indeed, results based on preliminary testing of the algorithm suggest that pointer variables typically do one of two things: 1) they shift between nodes whose priority numbers are identical, thus causing no reordering to occur; or 2) they shift between a fixed set of nodes, and once they have shifted to the highest number node, reordering never occurs again. The former case arises frequently in simulations where an object is typically moving between independent but fairly similar objects that have roughly the same number of constraints and the latter case arises frequently in menus where the last item has the constraints with the highest priority number (because it is the last item laid out). Thus in practice, the algorithm appears to fairly rapidly quiesce to a state where very few reorderings occur during constraint evaluation.

Other Implementation Issues

Each time a constraint is evaluated, its value is cached so that the next time the constraint's value is requested, the constraint will not be reevaluated unless one of its parameters has changed. Similarly the values of paths can be cached to improve efficiency. For example, in the labeled box example presented in Section 3.2, the label accessed the position of the box using the path `(self.parent.box)`. The first time this path is evaluated, the constraint solver can cache the resulting pointer to the box, so that as long as the variables comprising the path do not change, the constraint behaves as a direct reference constraint. The variables on this path still maintain dependency pointers to the constraint, so that if one of these variables changes, the path can be reevaluated and a new value cached for it.

Another implementation issue that arises is what to do with constraints containing variables that are nil or which reference deleted objects. The two options considered in Garnet were 1) to destroy the constraint, keeping its previously computed value; or 2) to keep the constraint and return its previously computed value. Under option two, the constraint would again be evaluated once all its variables point at valid objects. We settled on the second option since, in many cases, the constraint will be used again. For example, feedback objects that are invisible may have their `obj-over` variables set to nil, yet the constraints should be maintained so that they can correctly position the feedback object once it is made visible and its `obj-over` variable is set to a newly selected item.

6 Status and Future Work

Indirect reference constraints have been completely implemented at a very low level in Garnet. Every layer in Garnet is implemented on top of the constraint system using indirect reference constraints, except for the lowest-level untyped object system. This includes the graphical object system, the handling of the input, and all the widget libraries. In addition, Garnet has approximately 150 users who have used indirect reference constraints to generate hundreds of applications.

Garnet currently uses lazy evaluation and a modified user-controlled version of caching that evaluates a path the first time the constraint is evaluated and then ignores it if the user assures the constraint solver that the path will never change. On a SUN Sparcstation 1+ running Lucid Common Lisp, an indirect reference to an object through a variable (e.g., `self.obj-over.left`) requires 170 microseconds, whereas a direct reference (e.g., `menu-item1.left`) or a reference that uses a cached path requires 54 microseconds. If a constraint does not have to be reevaluated, its previously computed value can be accessed in 19 microseconds, regardless of whether it is a direct reference or indirect reference constraint. Garnet's constraint solver can solve indirect reference constraints quickly enough to allow feedback objects to track the mouse in real time or to perform smooth, realtime animations, even in large, constraint-based applications. For example, the Lapidary interactive design tool [10] consists of 16000 lines of Lisp code and 23500 constraints, all of which are indirect reference constraints, and is fast enough to provide instantaneous feedback to the user.

We have a working version of an eager evaluator that we believe is more efficient than the current lazy evaluator and which should be implemented in Garnet in the near future. We also have a design for two-way indirect reference constraint systems. Finally, we are examining graphical means of tracing these constraints so that designers can debug them more easily [12].

7 Conclusions and Future Work

Indirect reference constraints allow procedural abstraction to be added to constraint systems. This significantly extends the potential uses of constraints in interactive applications by allowing constraints to express the dynamic behavior that occurs inside an application's window. These constraints can be used to specify animations and feedback that operate over dynamic sets of objects, implement copying and instancing of structured objects in prototype-instance systems, simplify the creation of prototype objects from example objects in demonstrational systems, and abstractly specify layouts. In addition, their programming style is simpler and more effective than conventional constraints, they improve the efficiency of applications, and they decrease an application's storage demands. Because of their flexibility and ease of use, indirect reference constraints have permitted a comprehensive user interface toolkit to be built for the first time on top of a constraint system. This represents an important step toward the development of a general-purpose, constraint-based, interactive programming language.

References

1. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics 5*, 2 (April 1986), 142-172.

2. Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics 5*, 4 (Oct. 1986), 345-374.

3. Bjorn N. Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. OOPSLA/ECOOP'90 Conference Proceedings, 1990, pp. 77-88.

4. R. Hoover. *Incremental Graph Evaluation*. Ph.D. Th., Department of Computer Science, Cornell University, Ithaca, NY, 1987.

5. Scott E. Hudson. Graphical Specification of Flexible User Interface Displays. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 105-114.

6. Scott E. Hudson. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. Tech. Rept. TR89-12, The University of Arizona, 1989.

7. Scott E. Hudson. An Enhanced Spreadsheet Model for User Interface Specification. Tech. Rept. TR90-33, The University of Arizona, 1990.

8. J. Jaffar and J. Lassez. Constraint Logic Programming. Proceedings of the Principles of Programming Languages Conference, ACM, Munich, Germany, Jan., 1987, pp. 111-119.

9. Brad A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

10. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 95-104.

11. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

12. Brad A. Myers. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 243-249.

13. T. Reps, T. Teitelbaum, and A. Demers. "Incremental Context-Dependent Analysis for Language-Based Editors". *ACM TOPLAS 5*, 3 (July 1983), 449-477.

14. V. A. Saraswat. *Concurrent Constraint Programming Languages*. Ph.D. Th., School of Computer Science, CMU, Pittsburgh, PA, 1989.

15. Guy L. Steele, Jr. *The Definition and Implementation of A Computer Programming Language based on Constraints*. Ph.D. Th., Department of Computer Science, MIT, Boston, MA, 1980.

16. Ivan E. Sutherland. SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.

17. Pedro A. Szekely and Brad A. Myers. "A User Interface Toolkit Based on Graphical Objects and Constraints". *Sigplan Notices 23*, 11 (Nov. 1988), 36-45. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.

18. Pedro Szekely. Template-Based Mapping of Application Data to Interactive Displays. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 1-9.

19. Brad T. Vander Zanden. Constraint Grammars--A New Model for Specifying Graphical Applications. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 325-330.

20. Brad Vander Zanden and Brad A. Myers. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'90, Seattle, WA, April, 1990, pp. 27-34.