# SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction

*Michael Sannella*
Department of Computer Science
and Engineering, FR-35
University of Washington
Seattle, Washington 98195
E-mail: sannella@cs.washington.edu

**ABSTRACT**

Many user interface toolkits use constraint solvers to maintain geometric relationships between graphic objects, or to connect the graphics to the application data structures. One efficient and flexible technique for maintaining constraints is multi-way local propagation, where constraints are represented by sets of *method* procedures. To satisfy a set of constraints, a local propagation solver executes one method from each constraint.

SkyBlue is an incremental constraint solver that uses local propagation to maintain a set of constraints as individual constraints are added and removed. If all of the constraints cannot be satisfied, SkyBlue leaves weaker constraints unsatisfied in order to satisfy stronger constraints (maintaining a constraint hierarchy). SkyBlue is a more general successor to the DeltaBlue algorithm that satisfies cycles of methods by calling external cycle solvers and supports multi-output methods. These features make SkyBlue more useful for constructing user interfaces, since cycles of constraints can occur frequently in user interface applications and multi-output methods are necessary to represent some useful constraints. This paper discusses some of the applications that use SkyBlue, presents times for some user interface benchmarks and describes the SkyBlue algorithm in detail.

**KEYWORDS:** SkyBlue, constraints, local propagation, constraint hierarchies, user interface implementation.

## 1 INTRODUCTION

User interface toolkits can use constraint solvers to implement the connection between application data and a display of that data, to maintain consistency among multiple views of data, and to establish geometric relationships among graphical objects. By giving the solver responsibility for maintaining the various relationships in a user interface, the programmer is freed from the tedious and error-prone task of maintaining these relationships by hand, making it easier to develop and maintain complex graphical user interfaces. Many user interface development systems have provided integrated constraint solvers, including Garnet [10], Rendezvous [8], and ThingLab II [9]. References [2, 6] contain additional references to constraint-based systems.

One efficient and flexible technique for maintaining constraints is *multi-way local propagation*. In this technique, each constraint is represented by a set of *method* procedures that read the values of some of the constrained variables, and calculate values for the remaining constrained variables that satisfy the constraint. A set of such constraints can be maintained by a constraint solver that chooses one *selected method* for each constraint so that no variable is set by more than one selected method (i.e., there are no *method conflicts*). If there are no cycles in the selected methods, the solver can sort and execute them to satisfy all of the constraints. For example, given the constraint $A + B = C$ (represented by three methods $C := A + B$, $A := C - B$, and $B := C - A$) and the constraint $B + C = D$ (represented by three analogous methods), the two constraints could be satisfied by executing the methods $C := A + B$ and $D := B + C$ in this order. If there is a cycle among the selected methods (such as the cycle formed by the two methods $C := A + B$ and $B := D - C$) then local propagation may not be able to satisfy the constraints.

For a given set of constraints, there may be multiple ways to select methods to satisfy the constraints. It is also possible that there is no way for the solver to select methods for all of the constraints so there are no method conflicts. The theory of constraint hierarchies [2] offers a way to control the behavior of a constraint solver in these situations. Given a constraint hierarchy, a set of constraints where each constraint has an associated *strength*, a constraint solver can leave weaker constraints unsatisfied in order to satisfy stronger constraints.

To appear: Proceedings UIST '94.

The DeltaBlue algorithm is an incremental algorithm for maintaining constraint hierarchies using local propagation [9, 15]. The ThingLab II user interface development environment was based on DeltaBlue, demonstrating its feasibility for constructing user interfaces [9]. However, DeltaBlue has two significant limitations: cycles in the graph of constraints and variables are prohibited (if a cycle is found, the cycle is broken by removing a constraint), and constraint methods can only have one output variable. The SkyBlue algorithm presented in this paper is a successor to the DeltaBlue algorithm that supports cycles and multi-output methods [12, 13].

Cycles of constraints can occur frequently in user interface applications, particular when geometric constraints are created among graphic objects. DeltaBlue's prohibition of cycles places an undue burden on the programmer who has to worry about inadvertently introducing cycles. Several local propagation constraint solvers handle cycles by executing the methods "once around the cycle," but this technique is unreliable—the cycle constraints may or may not be satisfied, and the programmer has to examine the entire cycle to determine which is the case. SkyBlue maintains the constraints in a cycle by passing them to specialized *cycle solvers*. For example, if all of the constraints in a cycle are linear equations, our SkyBlue implementation calls a solver based on Gaussian elimination to satisfy the constraints around the cycle, and uses local propagation to satisfy the rest of the constraints. If the available cycle solvers can't satisfy the constraints in a cycle, the variables are marked invalid, so the programmer can tell when the cycle constraints are not satisfied.

SkyBlue supports constraints with *multi-output methods* that set the values of multiple output variables. Although most constraints in user interface applications can be represented with single-output methods, multi-output methods are necessary to represent some useful constraints. For example, suppose the variables $X$ and $Y$ represent the Cartesian coordinates of a point, and the variables $\rho$ and $\theta$ represent the polar coordinates of the same point. To keep these two representations consistent, one can define a constraint with one method calculating $X$ and $Y$ from $\rho$ and $\theta$, and another method calculating $\rho$ and $\theta$ from $X$ and $Y$. Multi-output methods are also useful for accessing the elements of compound data structures. For example, one could unpack a compound Cartesian point object into two variables using a constraint with methods $(X, Y) := (Point.X, Point.Y)$ and $Point := CreatePoint(X, Y)$.

Support for cycles and multi-output methods introduces a performance issue. Given a set of constraints with no cycles or multi-output methods, the worst-case time complexity of SkyBlue for adding or removing a constraint is linear in the number of constraints in the set (assuming that the number of constraint strengths, the number of variables per constraint, and the number of methods per constraint are all bounded by small constants). However, it has been proven that supporting cycles and multi-output methods is NP-complete [9]. Special examples have been constructed where the time for SkyBlue to add or remove a particular constraint is exponential in the number of constraints in the set. These test cases are highly unusual, and it is unlikely that similar sets of constraints would be constructed in a real application. In actual use, SkyBlue typically examines only a small subset of the constraints when a constraint is added or removed, and the performance is sub-linear in the number of constraints in the set.

The following sections list some of the applications built using SkyBlue and describes user interface benchmarks comparing SkyBlue to another solver. The rest of this paper presents the details of the SkyBlue algorithm. The author's dissertation describes SkyBlue in more detail [12]. Contact the author for information about implementations of SkyBlue.

## 2 SKYBLUE APPLICATIONS

SkyBlue has been used in a number of applications produced by different research groups:

- The Multi-Garnet package [14] uses SkyBlue to add support for multi-way constraints and constraint hierarchies to Garnet [10], a user interface toolkit built on Common Lisp and X windows. Multi-Garnet constraints support many of the useful features of Garnet's one-way constraints (formulas), including indirect references to constrained object slots though a series of other slots and inheritance of constraints in Garnet's prototype-based object system. Multi-Garnet allows the SkyBlue constraint solver to coexist with the Garnet constraint solver. It is possible to run existing Garnet programs without change and build Multi-Garnet applications using Garnet's library of widgets.
- The CoolDraw constraint-based drawing program [5] uses an extended version of SkyBlue to maintain geometric relationships between graphic objects in a two-dimensional plane.
- TBAG is a toolkit for creating interactive 3D graphics [3] that uses SkyBlue to maintain relationships between time-varying properties of graphic objects such as their positions and the derivatives of their positions.
- The VB2 virtual reality system [7] uses SkyBlue to maintain connections between 3D input devices and objects in the virtual world, and to attach virtual tools to objects that the user is editing. These constraints are added and removed as the user manipulates different objects in the virtual world.
- The Kaleidoscope language [4] integrates constraints and imperative, object-oriented programming. The current implementation of this language (Kaleidoscope'93) uses SkyBlue to maintain primitive constraints.
- The Pika simulation system [1] constructs simulations in domains such as electronics or thermodynamics by collecting algebraic and differential equations representing relationships between object attributes. Pika uses SkyBlue to process the equations for a numerical integrator that maintains the equations during the simulation.

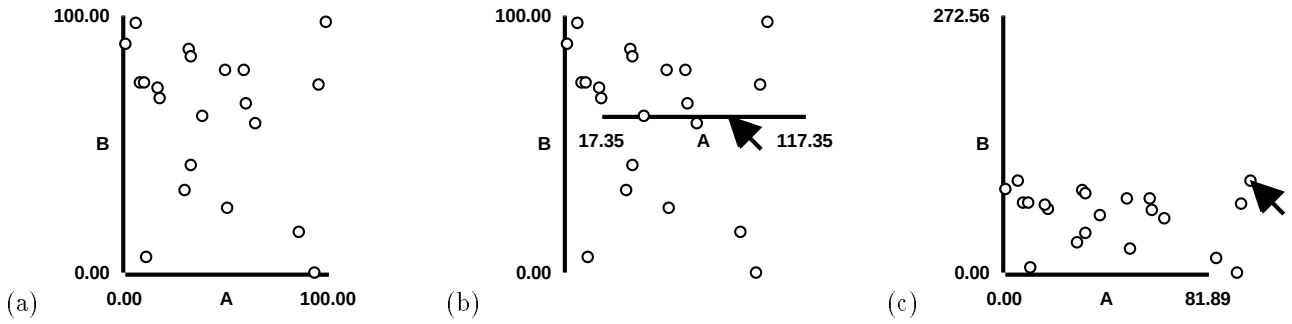



Figure 1: Three views of a scatterplot built in Multi-Garnet. (a) The initial scatterplot. (b) After moving the X-axis. (c) After scaling the point cloud by moving a point.

Figure 1 shows three pictures of a graphic user interface constructed in Multi-Garnet: A scatterplot displaying a set of points. SkyBlue constraints are used to specify the relationship between the screen position of each point, the corresponding data value, and the positions and range numbers of the axes. As the points and axes are moved with the mouse, SkyBlue maintains the constraints so that the plot continues to display the same data. When an object is moved there may be many possible ways to satisfy the constraints. The scatterplot program specifies the behavior of the user interface by adding *stay* constraints to variables that should not be changed by the solver. For example, in Figure 1b stays are added to the positions of the data points, so only way to maintain the constraints when the axis is moved is to change the axis range numbers. Each interaction is implemented by adding a different set of stay constraints.

The scatterplot program exploits many of the features of SkyBlue. SkyBlue resatisfies the constraints quickly enough to allow continuous interaction. The constraints operate in multiple directions during different interactions. Some of the constraints have multi-output methods. While this application could have been written entirely in Garnet using one-way constraints, for each different interaction mode the programmer would have had to enable exactly the right set of one-way constraints to propagate the values correctly. In contrast, the Multi-Garnet scatterplot program declares the relationships between the scatterplot elements once, and SkyBlue determines which methods to execute to maintain the constraints (influenced by the stay constraints).

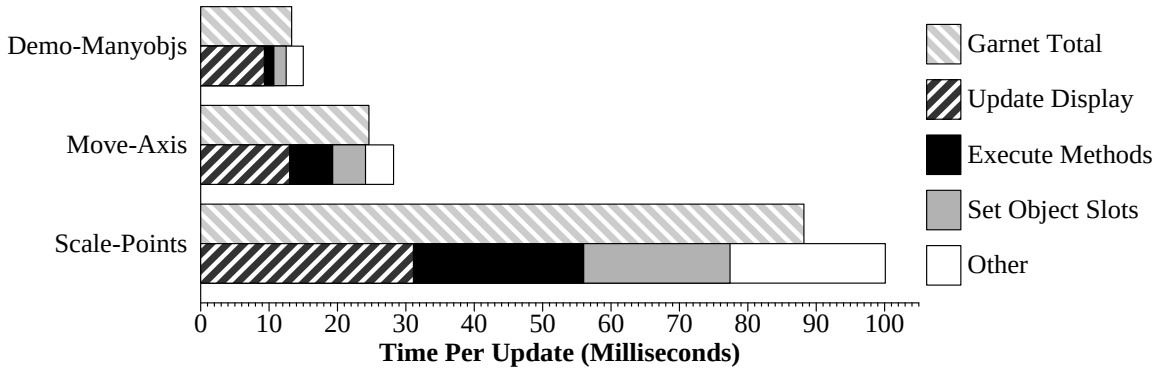
## 3 GARNET AND MULTI-GARNET PERFORMANCE

In order to compare the performance of Multi-Garnet to Garnet, several benchmark programs were implemented in Garnet and Multi-Garnet. Demo-Manyobjs displays a chain of boxes connected by lines positioned with one-way constraints. The Demo-Manyobjs benchmark measures the time to move a box, resatisfy the constraints and reposition the connecting lines. Each update executes 16 constraint methods (16 formulas in the Garnet version), mostly performing fast integer arithmetic, and redisplays a single box and two lines. The Move-Axis benchmark measures the time to move the scatterplot X-axis once in Figure 1b. Each update executes 22 methods (23 Garnet formulas), including methods that perform floating-point arithmetic and string formatting, and redisplays the axis line, the range numbers, and any points overlapping the axis. There are more formulas than methods because one multi-output method performs the same calculation as two formulas. The Scale-Points benchmark measures the time to scale the point cloud by moving a point once in Figure 1c. Each update executes 112 methods (178 Garnet formulas), including many methods performing floating-point arithmetic to calculate the position of each scaled point, and redisplays all of the points.

The Garnet versions of the scatterplot benchmarks are implemented with one-way constraints, so they only support a small part of the functionality of the Multi-Garnet versions. For example, the Garnet version of the Scale-Points benchmark only allows one particular point to be moved to scale the point cloud, whereas any point could be moved in the Multi-Garnet version.

Figure 2 presents times measured running the benchmarks using Garnet v2.2 and Franz Allegro Common Lisp v4.1 on a Sun Microsystems SPARCstation IPX. The total Garnet and Multi-Garnet numbers indicate the time (in milliseconds) to perform the sequence: move a constrained element, resatisfy the constraints, and update the display. This figure shows that the Multi-Garnet versions of the benchmark programs are somewhat slower than the Garnet ones. However, it should be emphasized that Multi-Garnet can do much more than Garnet. The important question is whether the difference in execution time is significant.

The Multi-Garnet benchmark times are divided into the portions of the total time spent updating the graphics, executing the method procedures, and updating the constrained Garnet object slots with the new values (an expensive operation which involves running demons associated with the objects). It is difficult to directly measure the components of the Garnet benchmark times, since Garnet intertwines the different operations, but the Garnet versions of the benchmarks should use approximately the same amount of time to perform the first three components.




| | *Garnet* | *Multi-Garnet* | *Update Display* | *Execute Methods* | *Set Object Slots* | *Other* |
|---|---|---|---|---|---|---|
| Demo-Manyobjs | 13.3 | 15.0 | 9.3 | 1.4 | 1.8 | 2.5 |
| Move-Axis | 24.6 | 28.2 | 13.0 | 6.3 | 4.8 | 4.1 |
| Scale-Points | 88.2 | 100.1 | 31.1 | 24.9 | 21.4 | 22.7 |

Figure 2: Garnet and Multi-Garnet benchmark timings. All times are in milliseconds.

The remainder of the Multi-Garnet time ("other") is divided between sorting the selected methods to be executed and performing other tasks that integrate SkyBlue into the Garnet system. This time could be reduced by extracting and reusing a *plan* containing the sorted list of methods to execute. The Multi-Garnet benchmark figures do not include the time to add the original constraint to inject the mouse positions into the network—this time is dominated by the time to repeatedly execute the methods and update the display when dragging the constrained object.

The graphics redisplay and method execution time would be essentially the same no matter how the application was implemented, whether a constraint solver was used or the relationships were maintained explicitly (integrating the method procedures into an imperative program). This is typical in interactive systems using local propagation, and suggests that constraint solvers such as SkyBlue can be used to construct interactive systems without significantly impacting performance.

## 4 THE SKYBLUE ALGORITHM

Externally, SkyBlue is similar to the older DeltaBlue algorithm. Both algorithms are called using the same entries, and they represent constraints and variables in similar ways. However, support for multi-output methods and cycles requires the SkyBlue implementation to be substantially different from DeltaBlue. The major differences between the implementations of DeltaBlue and SkyBlue are: (1) SkyBlue uses a backtracking search when selecting methods to execute (Section 5), (2) SkyBlue generalizes the concept of *walkabout strengths* used in DeltaBlue (Section 7), and (3) SkyBlue provides a mechanism for calling external constraint solvers to satisfy constraints in cycles (Section 8).

SkyBlue has two entries, `add_constraint` to add a constraint to the set of constraints that SkyBlue is maintaining, and `remove_constraint` to remove a constraint. When either of these is called, SkyBlue determines which constraint methods to use to maintain the new set of constraints, and executes them. SkyBlue maintains a data structure known as a *method graph* (or *mgraph*) consisting of a set of constraints along with the selected method for each of the constraints. If a constraint has a selected method in the mgraph, it is *enforced*, otherwise it is *unenforced*. Enforcing (unenforcing) a constraint means assigning a selected method (no selected method) to a constraint. When a constraint is added or removed, SkyBlue incrementally updates the mgraph by enforcing stronger constraints and unenforcing weaker constraints, and sorts and executes the selected methods, calling cycle solvers to handle cycles.

Formally, SkyBlue constructs a *Method-Graph-Better* (or *MGB*) mgraph, where *mg* is an MGB mgraph if it has no method conflicts and for each unenforced constraint *cn* in *mg* there exists no conflict-free mgraph for the same constraints where *cn* is enforced and all of the enforced constraints of *mg* with the same or stronger strength as *cn* are enforced. By constructing MGB mgraphs, SkyBlue ensures that weaker constraints are left unenforced if necessary to enforce stronger constraints. However, weaker constraints can influence which selected methods are used to enforce stronger constraints. MGB mgraphs are similar to the Locally-Graph-Better graphs constructed by DeltaBlue [9]. The variable values produced by executing the enforced constraints are usually (but not always) Locally-Predicate-Better solutions as defined by the theory of constraint hierarchies [2, 12]. SkyBlue can handle any number of different constraint strengths. Examples in this paper use the strengths *max*, *strong*, *medium*, *weak* (strongest to weakest).

As an example, consider the mgraphs in Figure 3. Black boxes are constraints, white circles are variables, and arrows specify the outputs of the selected method for

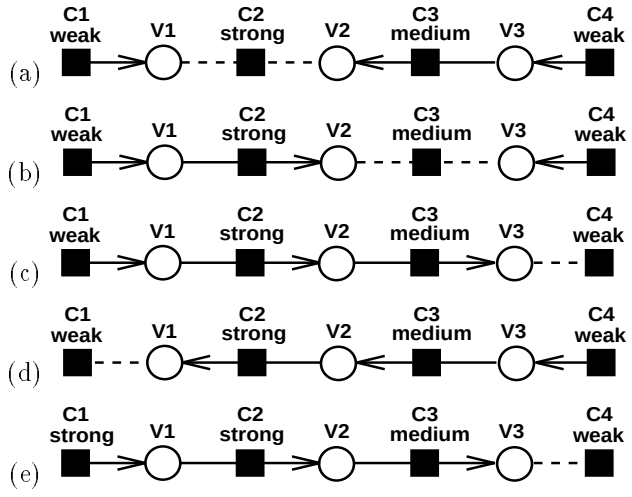


Figure 3: MGB and non-MGB Method Graphs.

the constraint. Dashed lines indicate unenforced constraints. Figure 3a is not an MGB mgraph because the *strong* constraint $C2$ could be enforced by choosing the method that outputs to $V2$ and unenforcing the *medium* constraint $C3$, producing Figure 3b. Actually, this mgraph is not MGB either since $C3$ could be enforced by unenforcing $C4$, producing Figure 3c. This mgraph is MGB since the only unenforced constraint ($C4$) cannot be enforced without introducing a method conflict or unenforcing equal-strength or stronger constraints. There may be multiple MGB mgraphs for a given set of constraints. Figure 3d shows another MGB mgraph for the same constraints. Given these constraints, SkyBlue could construct either one of these mgraphs. The constraint strengths could be modified to favor one alternative over the other. For example, if the strength of $C1$ were changed to *strong*, the only MGB mgraph would be the one in Figure 3e.

## 5 BUILDING METHOD VINES

SkyBlue updates the mgraph to be MGB by repeatedly calling the procedure `build_mvine`, the heart of the SkyBlue algorithm. This procedure tries to enforce an unenforced constraint `root` by selecting a method for `root`, unenforcing constraints that are weaker than `root`, and switching methods for other enforced constraints. If it is unable to construct a conflict-free mgraph where `root` is enforced, it leaves the mgraph unchanged. It has been proven that if none of the unenforced constraints are *mvine-enforcible* (i.e., can be enforced by `build_mvine`), then the mgraph must be an MGB mgraph [12].

The procedure `build_mvine` tries to enforce `root` by the following process: `root` is enforced with one of its methods. If this method has a method conflict with the selected methods of other enforced constraints, these conflicting constraints are unenforced. If a conflicting constraint is equal-strength or stronger than `root`, then it is enforced with a different selected method. If the new selected methods for the conflicting constraints conflict with yet other enforced constraints, these new conflicting constraints are handled in the same way, and so on. This process extends through the mgraph, building a "vine" of newly-chosen selected methods growing from `root`.

If `build_mvine` processes a conflicting constraint that is equal-strength or stronger than `root` (so it must be enforced), and all of the methods of this constraint conflict with other newly-selected methods, then the mvine construction process backtracks: Previously-processed constraints are unenforced and the mvine is extended using other selected methods for these constraints. This backtracking process may unwind the search all the way to the beginning, choosing another selected method for `root`. If no method can be chosen for `root` that allows a complete conflict-free mvine to be constructed, then `build_mvine` fails and the mgraph is not changed.

Figure 4 shows an example of constructing an mvine to enforce $C1$. The newly-selected methods are drawn with thicker lines. The arrows below $C2$ indicate $C2$'s possible methods—all other constraints have a single-output method for each variable. Suppose SkyBlue starts with the mgraph of Figure 4a. First, $C1$ is enforced with its only method so it determines $V1$, and $C2$ is unenforced and added to the mvine (4b). Since $C2$ is stronger than the root constraint $C1$, it will have to be enforced in the mvine. The method that outputs to $V1$ conflicts with $C1$, so $C2$ is enforced with its other method, and $C4$ is unenforced and added to the mvine (4c). Since $C4$ is stronger than the root, it will have to be enforced. Suppose SkyBlue enforces $C4$ with the method that sets $V4$, unenforcing $C3$ (4d). Since $C3$ is stronger than the root, it will have to be enforced. Both of its methods conflict with other constraints in the mvine ($C2$ and $C4$), so SkyBlue backtracks, removing $C3$ from the mvine, enforcing it with its original method, and unenforcing $C4$. It is still necessary to enforce $C4$, so SkyBlue enforces $C4$ with another method, unenforcing $C5$ and adding it to the mvine (4e). Now, $C5$ is the only unenforced constraint in the mvine, and it is weaker than the root constraint, so SkyBlue has successfully constructed an mvine.

The backtracking search in `build_mvine` is pruned by the use of *walkbounds* (Section 7) associated with the variables that allow `build_mvine` to predict that particular methods cannot be used to enforce an mvine constraint. If there are no cycles or constraints with multi-output methods in the mgraph, then `build_mvine` will never backtrack. In the example of Figure 4 (containing a multi-output method), walkbounds do not affect the search.

Figure 5 contains pseudocode for an implementation of `build_mvine`. The recursive procedure `extend_mvine` is called with the root constraint and the set of constraints in the mvine. This set initially contains just the unenforced root constraint, but it is extended on recursive calls with conflicting constraints that have been added to the mvine.

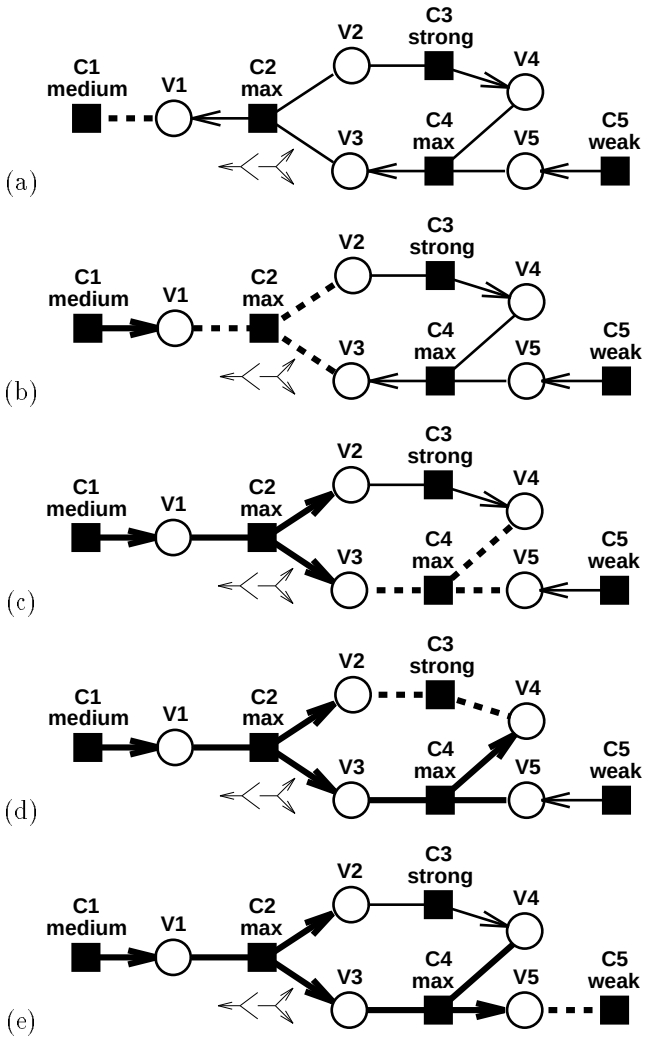



Figure 4: Constructing an mvine.

If there are no unenforced constraints in the mvine with strength equal or stronger than the root constraint, then a complete mvine has been successfully constructed, and `extend_mvine` returns true. Otherwise, `extend_mvine` chooses a constraint from the mvine that needs to be enforced (`new_cn`), selects a method for it, unenforces any conflicting constraints, and calls itself recursively (adding the unenforced conflicting constraints to the mvine). Note that this procedure never enforces any constraint with a method that conflicts with other enforced constraints in the mvine.

If the recursive call returns true, indicating that the rest of the mvine constraints have been handled successfully, then `extend_mvine` returns true itself. If the recursive call returns false, then it must not be possible to construct the rest of the mvine. In this case, the selected methods for `new_cn` and the conflicting constraints are reset, and the next method for `new_cn` is examined.

There are three situations where a particular method cannot be used to enforce a constraint: (1) The method conflicts with an enforced constraint in the mvine. (2) The walkbounds of the method's outputs are too strong as determined by `check_walks` (described in Section 7). In the example of Figure 4, `check_walks` always returns true. (3) After setting the selected method, the recursive call to `extend_mvine` cannot successfully construct the rest of the mvine. If none of a constraint's methods is acceptable, then `extend_mvine` leaves the constraint unenforced, and returns false, backtracking.

```
build_mvine(root): boolean
   Return extend_mvine(root, {root})

extend_mvine(root, mvine_cns): boolean
   new_cn := choose any unenforced constraint
             in mvine_cns with strength equal
             or stronger than root.strength
   If there is no such constraint then
      Return true
   For new_mt in new_cn.methods do
      conflict_cns := all constraints whose
         selected method conflicts with new_mt
      If ( conflict_cns ∩ mvine_cns = {} ) and
        check_walks(new_cn, new_mt, root) then
          ;; unenforce conflict_cns, enforce new_cn
          For cn in conflict_cns do
             cn.original_mt := cn.selected_mt
             cn.selected_mt := nil
          new_cn.selected_mt := new_mt
          ;; try constructing the rest of the mvine
          next_cns := mvine_cns ∪ conflict_cns
          If extend_mvine(root, next_cns) then
             Return true
          ;; can't build rest of mvine, undo changes
          new_cn.selected_mt := nil
          For cn in conflict_cns do
             cn.selected_mt := cn.original_mt
   ;; new_cn can't be enforced; backtrack
   Return false
```

Figure 5: Pseudocode for building an mvine.

## 6 UPDATING THE MGB METHOD GRAPH

SkyBlue could update the mgraph by calling `build-mvine` repeatedly on *all* of the unenforced constraints, until none of them can be enforced. However, often it is not necessary to test all of the unenforced constraints. For example, suppose that the constraints define two completely disjoint networks of constraints and variables. If a constraint is removed from one of the networks, this cannot affect whether unenforced constraints in the other network are mvine-enforcible.

SkyBlue manages the unenforced constraints by maintaining the set `*pmec*` (containing the Possibly Mvine-Enforcible Constraints). Every time that the mgraph is changed by adding or removing a constraint or applying an mvine, this set is updated to include any newly mvine-enforcible constraints. When an unenforced constraint in `*pmec*` is enforced (by building an mvine), or when it is determined that it is not mvine-enforcible

(when `build_mvine` fails), it is removed from `*pmec*`. When `*pmec*` is empty, then none of the unenforced constraints in the mgraph are mvine-enforcible, and the mgraph must be MGB.

Now, the problem has been transformed to that of updating `*pmec*` whenever the mgraph is changed so that it always contains all mvine-enforcible constraints. SkyBlue uses several techniques to update `*pmec*` when the mgraph is changed. It has been proven that these techniques will collect all of the newly mvine-enforcible constraints (in addition to some constraints that are not mvine-enforcible) whenever the mgraph is changed [12].

- *The Collect Local Unenforced Technique.* Whenever an unenforced constraint is added to the mgraph, it must be added to `*pmec*`. Whenever the mgraph is changed in any other way, the only constraints in the mgraph that need to be added to `*pmec*` are the unenforced constraints whose variables include redetermined variables, or variables downstream of the redetermined variables. The *redetermined variables* are defined as those variables whose determining method in the mgraph has been changed. Downstream variables are found by following the links in the directed graph formed by the selected methods (with directed links from a selected method to its outputs, and from a variable to the selected methods that input it). Note that no constraints have to be added to `*pmec*` when an unenforced constraint is removed, since no variables are redetermined.
- *The Removed Constraint Strength Technique.* Whenever an enforced constraint is removed from the mgraph, only unenforced constraints with the same strength or weaker than the removed constraint need to be added to `*pmec*`.
- *The Mvine Root Strength Technique.* Whenever an mvine is successfully built, only unenforced constraints that are strictly weaker than the mvine root constraint need to be added to `*pmec*`.

Figure 6 contains pseudocode for adding or removing a constraint from the mgraph, updating the mgraph to be MGB, updating the variable walkbounds (Section 7) and executing the selected methods (Section 8). This pseudocode incorporates all of the techniques mentioned above to update `*pmec*`. Note that the procedure `update_method_graph` always removes and processes the strongest constraint in `*pmec*`, and only adds constraints strictly weaker than the root of the mvine when one is constructed, so no constraint will be processed more than once during a single call to this procedure.

```
add_constraint(cn)
   add cn to the method graph (unenforced)
   *pmec* := {cn}
   update_method_graph()
   execute_selected_methods()

remove_constraint(cn)
   *pmec* := {}
   If cn is enforced then
      out_vars := cn.selected_mt.outputs
      add to *pmec* all unenforced constraints
               with variables in or downstream
               of out_vars and strength equal
               to or weaker than cn.strength
      cn.selected_mt := nil
      update_walkbounds(old_outputs)
   remove cn from the method graph
   update_method_graph()
   execute_selected_methods()

update_method_graph()
   Loop until *pmec*={} do
      cn_to_enforce := choose the strongest
             unenforced constraint in *pmec*
      *pmec* := *pmec* - {cn_to_enforce}
      If build_mvine(cn_to_enforce) then
         redetermined_vars := all variables
               redetermined by the mvine
         add to *pmec* all unenforced
              constraints with variables in or
              downstream of redetermined_vars
              and strength weaker than
              cn_to_enforce.strength
         update_walkbounds(redetermined_vars)
```

Figure 6: Pseudocode for adding or removing a constraint and updating the method graph to be MGB.

## 7 CALCULATING AND USING WALKBOUNDS

Consider the situation where `build_mvine` is about to choose a new selected method for $C1$ that determines $V1$, currently determined by the selected method of $C2$ (Figure 7a). In order to construct a complete conflict-free mvine, `build_mvine` changes the mgraph so that $C2$ no longer determines $V1$ by unenforcing $C2$ (7b), adding it to the mvine, and trying to enforce it with another method (if it is not weaker than the root constraint).

In this situation, `build_mvine` assumes that it will be possible to change the mgraph so that $C2$ no longer determines $V1$, without unenforcing any constraints that are equal-strength or stronger than the mvine root. If this assumption is wrong, it will eventually backtrack to this point, and choose another selected method for $C1$. If `build_mvine` could predict that this assumption is wrong, then it could immediately reject $C1$'s new selected method without trying to extend the mvine. This would reduce the amount of searching done while building an mvine.

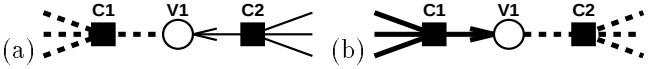



Figure 7: Enforcing $C1$ and unenforcing $C2$.

The DeltaBlue algorithm introduced the idea of associating a *walkabout strength* with each variable that specifies the strength of the strongest constraint that would have to be unenforced to allow a different method to determine that variable. If a variable is undetermined, or

the mgraph could be modified so the variable is no longer determined without unenforcing any constraints, then its walkabout strength is defined as *min*, which is a special strength weaker than any constraint. For DeltaBlue constraints (no cycles or multi-output methods) it is not difficult to calculate these strengths and incrementally update them whenever the mgraph is changed.

If an mgraph contains cycles and multi-output methods, it is hard to calculate walkabout strengths efficiently. SkyBlue calculates lower bounds to the walkabout strengths called *walkbounds*. Using the variable walkbounds, SkyBlue can reduce the amount of searching it performs, but cannot eliminate it completely. Figure 8 shows pseudocode for `check_walks`, called by `extend_mvine` to determine whether a method can be used to extend the mvine. This is only possible if all of the output variables of the new method have walkbounds weaker than the root of the mvine.

Figure 8 also shows pseudocode for `update_walkbounds`, which is called to update the walkbounds whenever the mgraph is modified. It has been proven that the only walkbounds that need to be updated after the mgraph is changed are the walkbounds of the redetermined variables and downstream variables [12]. This procedure takes all of the constraints that determine these variables, and collapses all constraint cycles into *collapsed cycle* methods (as described in Section 8). Then, the collapsed cycle methods and the other selected methods are topologically sorted and processed in order. Methods are passed to `set_walkbounds`, which sets the walkbounds of the outputs of an enforced constraint's selected method, given the walkbounds of the method's inputs. Collapsed cycle methods are processed by setting the walkbounds of all variables set by the cycle constraints *min* (a correct walkbound for any variable), and then passing each cycle constraint to `set_walkbounds`.

For each selected method output variable, `set_walkbounds` examines the alternate methods of the constraint that don't output to it (and thus could be used to enforce the constraint while leaving this variable undetermined). For each of these methods the strongest walkbound among the method's outputs is found (by calling `max_out`). If a selected method has multiple outputs, this may calculate different walkbounds for the different output variables.

Both `check_walks` and `max_out` include code to handle the special case where two methods for the same constraint output to the same variable. In `check_walks`, it is not necessary to check the walkbound of a variable that is output by the constraint's current selected method, since no additional constraints need to be unenforced to switch to another selected method that outputs to the same variable. Likewise, in `max_out` the walkbounds of any variables output by the constraint's original selected method can be ignored.

```
check_walks(cn, new_mt, root): boolean
   If cn = root then
      old_outputs := {}
   Else
      old_outputs := cn.original_mt.outputs
   For var in new_mt.outputs - old_outputs do
      If var.walkbound is equal or stronger
              than root.strength then
         Return false
   Return true

update_walkbounds(redetermined_vars)
   For var in redetermined_vars do
      If var is undetermined then
         var.walkbound := min
   downstream_cns := all enforced constraints
            that determine or are downstream
            of redetermined_vars
   cn_and_cycle_list := topological_sort(
              collapse_cycles(downstream_cns))
   For x in cn_and_cycle_list do
      If x is a constraint then
         set_walkbounds(x)
      ElseIf x is a collapsed cycle then
         update_walkbounds_in_cycle(x)

update_walkbounds_in_cycle(cycle)
   cycle_cns := the constraints in cycle
   For cn in cycle_cns do
      For var in cn.selected_mt.outputs do
         var.walkbound := min
   For cn in cycle_cns do
      set_walkbounds(cn)

set_walkbounds(cn)
   out_vars := cn.selected_mt.outputs
   For out_var in out_vars do
      new_walk := cn.strength
      For mt in cn.methods do
         If out_var ∉ mt.outputs then
            new_walk := the weaker of new_walk
                     and max_out(mt, out_vars)
      out_var.walkbound := new_walk

max_out(mt, current_outputs): Strength
   max_strength := min
   For var in mt.outputs do
      If var ∉ current_outputs then
         max_strength := the stronger of
               max_strength and var.walkbound
   Return max_strength
```

Figure 8: Pseudocode for using and recalculating walkbounds.

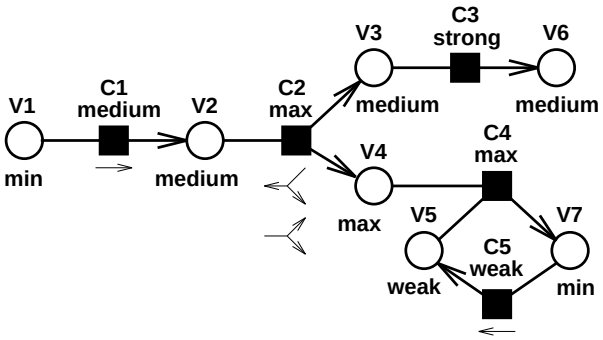



Figure 9: A method graph with walkbounds.

Figure 9 demonstrates how SkyBlue calculates walkbounds. The variables are labeled with walkbounds calculated by setting $V1$'s walkbound to *min*, processing $C1$–$C3$ by calling `set_walkbounds`, and handling the cycle by setting the walkbounds of $V5$ and $V7$ to *min* and calling `set_walkbounds` to process $C4$ and $C5$. Note that the walkbound of $V4$ is *max* because the unselected method for $C2$ also outputs to this variable. Also note that the walkbound of $V7$ would be set to *weak* rather than *min* if $C5$ had been processed before $C4$, but in either case these are correct walkbounds.

## 8 EXECUTING SELECTED METHODS

Figure 10 contains pseudocode that executes the selected methods in the mgraph to satisfy the enforced constraints. It is only necessary to execute "new" selected methods (ones that weren't selected the last time methods were executed) and selected methods downstream of those methods [12]. If there are no directed cycles in the mgraph, the selected methods are executed in topological order.

If there are directed cycles in the mgraph, it is not possible to topologically sort them. In this case, the cycles (actually, the strongly-connected components) are collapsed to produce *collapsed cycle* methods. For example, in Figure 11a, the selected methods for $C2$ and $C3$ form a directed cycle. These constraints are treated as if they were a single method that reads $V2$ and sets $V3$ and $V4$ to values that satisfy constraints $C2$ and $C3$ (Figure 11b). Once the cycles are collapsed, the regular methods and collapsed cycles are processed in topological order. Regular constraint methods are executed. When a cycle is encountered in the order, a series of external *cycle solvers* are called that try to find values for the cycle output variables to satisfy the constraints. For example, if all of the constraints are linear equations, a cycle solver incorporating a simultaneous linear equation solver could produce values that satisfy the constraints. After a cycle is solved, downstream methods and cycles are processed. Note that there is a difference between a cycle solver that returns that it cannot solve the cycle (as a linear equation solver would do when passed a cycle with non-linear constraints) and a cycle solver that returns that the cycle cannot be solved (as a linear equation solver would do

```
execute_selected_methods()
   exec_cns := all enforced constraints
     whose selected method has changed since
     execute_selected_methods was last called,
     and all downstream enforced constraints
   cn_and_cycle_list := topological_sort(
                    collapse_cycles(exec_cns))
   For x in cn_and_cycle_list do
      If x is a constraint then
         execute x.selected_mt
      ElseIf x is a collapsed cycle then
         call_cycle_solvers(x)

call_cycle_solvers(cycle): boolean
   For cycle_solver in *cycle_solvers* do
      call cycle_solver to solve cycle
      If cycle_solver solved cycle then
         Return true
      If cycle_solver determined that cycle
                         cannot be solved then
         Return false
    ;; none of the solvers could satisfy the cycle
   Return false
```

Figure 10: Pseudocode for executing the selected methods and calling cycle solvers.

when passed inconsistent linear equations). In the former case, `call_cycle_solvers` will try the rest of the cycle solvers, whereas in the latter case no other cycle solvers will be called.

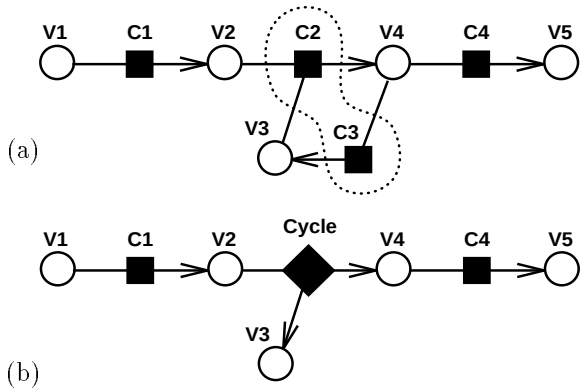



Figure 11: Collapsing a directed cycle of methods.

If none of the cycle solvers can solve a cycle, or a cycle solver determines that the cycle cannot be solved, the cycle variable values are left unchanged, and the downstream methods (and cycles) are not executed (this is not shown in the pseudocode). The variables downstream of the unsolved cycles are also marked, so the programmer can determine that the variable values do not satisfy the constraints.

## 9 CONCLUSIONS

This paper has described how SkyBlue can be used to maintain relationships in user interface construction, and

presented the details of the algorithm. There are many ways that SkyBlue could be improved, for example the procedure `call_cycle_solvers` could be modified to keep track of which cycle solvers are more successful at solving a given cycle, and use this information to call more promising cycle solvers first. One direction for future work is to extend SkyBlue to support inequalities. The CoolDraw system used an extended version of SkyBlue to support some inequalities [5].

Another area of active research is building debugging tools to help the programmer examine the constraint network, determine why a given solution is produced, and change the network to produce the desired solution. A system has been developed for interactively constructing graphical user interfaces based on constraints (maintained by SkyBlue), and debugging the constraint networks [11]. This debugging system uses a new algorithm for generating all of the MGB mgraphs for a set of constraints. This algorithm is the basis for a powerful debugging tool that allows the programmer to explore the different behaviors that can be produced by a set of constraints.


## ACKNOWLEDGEMENTS

Thanks to Alan Borning for useful comments on this paper. This work was supported in part by National Science Foundation grants IRI-9102938, IRI-9302249, and CCR-9402551, and by Academic Equipment Grants from Sun Microsystems.


## REFERENCES


1. Franz G. Amador, Adam Finkelstein, and Daniel S. Weld. Real-Time Self-Explanatory Simulation. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 562–567. AAAI Press/The MIT Press, July 1993.

2. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

3. Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. In *SIGGRAPH '94 Conference Proceedings*, pages 421–434, Orlando, Florida, July 1994. ACM. Also in *Computer Graphics* 28(2), July 1994.

4. Bjorn Freeman-Benson. *Constraint Imperative Programming.* PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1991. Published as UW CSE Technical Report 91-07-02.

5. Bjorn Freeman-Benson. Converting an Existing User Interface to Use Constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 207–215, Atlanta, Georgia, November 1993.

6. Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

7. Enrico Gobbetti and Jean-Francis Balaguer. VB2: An Architecture for Interaction in Synthetic Worlds. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 167–178, Atlanta, Georgia, November 1993.

8. Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The Rendezvous Architecture and Language for Constructing Multi-User Applications. *ACM Transactions on Computer-Human Interaction*, 1(2), 1994. To appear.

9. John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as UW CSE Technical Report 91-08-12.

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

11. Michael Sannella. Analyzing and Debugging Hierarchies of Multi-way Local Propagation Constraints. In Borning, editor, *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming.* Springer-Verlag, 1994. To appear.

12. Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces.* PhD thesis, Department of Computer Science and Engineering, University of Washington, 1994.

13. Michael Sannella. The SkyBlue Constraint Solver and Its Applications. In Saraswat and van Hentenryck, editors, *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming.* MIT Press, 1994. To appear.

14. Michael Sannella and Alan Borning. Multi-Garnet: Integrating Multi-Way Constraints with Garnet. Technical Report 92-07-01, Department of Computer Science and Engineering, University of Washington, September 1992.

15. Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.