View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-95-22

1995-01-01

User Interface Applications of a Multi-way Constraint Solver

T. Paul McCartney

Constraints are widely recognized as a useful tool for user interface constructino. Through constraints, relationships among user interface components can be defined declaratively, leaving the task of relationship management to a constraint solver. Multi-way constraint solvers supporting constraint hierarchies provide a means to specify preferential constraint relationships with a dynamically changing computation flow, making them especially well suited to interactive user interfaces. However, previous such constraint solvers lack the ability to enforce inequalities or to effectively handle cyclic constraint relationships. These deficiencies limit the problems that could be solved using a constraint-based approach. This paper presents a new algorithm... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

McCartney, T. Paul, "User Interface Applications of a Multi-way Constraint Solver" Report Number: WUCS-95-22 (1995). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/380

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/380

User Interface Applications of a Multi-way Constraint Solver

T. Paul McCartney

Complete Abstract:

Constraints are widely recognized as a useful tool for user interface constructino. Through constraints, relationships among user interface components can be defined declaratively, leaving the task of relationship management to a constraint solver. Multi-way constraint solvers supporting constraint hierarchies provide a means to specify preferential constraint relationships with a dynamically changing computation flow, making them especially well suited to interactive user interfaces. However, previous such constraint solvers lack the ability to enforce inequalities or to effectively handle cyclic constraint relationships. These deficiencies limit the problems that could be solved using a constraint-based approach. This paper presents a new algorithm called UltraBlue for solving hierarchies of multi-way constraints and discusses its application to the architecture of the EUPHORIA user interace management system. Contributions of UltraBlue include a value consistency mechanism for maintaining arbitrary assertoins (e.g., inequality relationships) and a cycle avoidance heuristic algorithm for eliminating cyclic constraint relationships. Cycles of constraints are resolved with respect to each constraint's relative strength, making it possible to construct acyclic constraint graphs that can be effectively solved, while preferring constraints of greater importance.

User Interface Applications of a Multi-way Constraint Solver

T. Paul McCartney

WUCS-95-22

October 1995

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

User Interface Applications of a Multi-way Constraint Solver

T. Paul McCartney
Department of Computer Science
Washington University
St. Louis, MO 63130
paul@cs.wustl.edu
http://www.cs.wustl.edu/~paul/

ABSTRACT

Constraints are widely recognized as a useful tool for user interface construction. Through constraints, relationships among user interface components can be defined declaratively, leaving the task of relationship management to a constraint solver. Multi-way constraint solvers supporting constraint hierarchies provide a means to specify preferential constraint relationships with a dynamically changing computation flow, making them especially well suited to interactive user interfaces. However, previous such constraint solvers lack the ability to enforce inequalities or to effectively handle cyclic constraint relationships. These deficiencies limit the problems that could be solved using a constraint-based approach.

This paper presents a new algorithm called UltraBlue *for solving hierarchies of multi-way constraints and discusses its application to the architecture of the* EUPHORIA *user interface management system. Contributions of UltraBlue include a value consistency mechanism for maintaining arbitrary assertions (e.g., inequality relationships) and a cycle avoidance heuristic algorithm for eliminating cyclic constraint relationships. Cycles of constraints are resolved with respect to each constraint's relative strength, making it possible to construct acyclic constraint graphs that can be efficiently solved, while preferring constraints of greater importance.*

KEYWORDS: constraints, constraint hierarchies, cycle avoidance, incremental constraint satisfaction, interactive techniques, multi-way constraints, user interface management system

1 INTRODUCTION

A *constraint* is a relationship to be maintained among a set of variables. For example, "A+B=C" represents the relationship that C is the sum of A and B. Constraints are widely recognized as a useful tool for user interface construction [1], [4], [10], [14], [20], [21], [23], [26]. Through constraints, relationships among user interface components and their applications can be defined declaratively, leaving the task of maintaining the relationships to a constraint solver, an algorithm that determines a plan for computing constrained values in a way that is consistent with the specified constraints. Constraints simplify the programmer's task in creating user interfaces and empower end-users to define sophisticated relationships without the need for programming.

Multi-way constraint solvers have proven to be useful in the development of user interfaces and other applications. However, previous such constraint solvers lacked the ability to express inequalities or to effectively handle cyclic constraint relationships. Cyclic constraint relationships pose many problems since, in general, the constraint relationships in a cycle cannot be satisfied efficiently (in many cases, it cannot be solved at all). This paper describes UltraBlue, a new constraint solver algorithm, and its application to the architecture and run-time system of the EUPHORIA user interface management system [6], [15], [17]. UltraBlue is an efficient incremental algorithm for solving hierarchies of multi-way, single-output, dataflow constraints using local propagation. That is, constraints have a dynamically changing computation direction, a hierarchy of enforcement preferences, and are represented using a dataflow graph structure with each constraint having a single output. Contributions of UltraBlue include a *value consistency* mechanism for maintaining arbitrary assertions (e.g., inequality relationships) and a *cycle*

avoidance heuristic algorithm for eliminating cyclic constraint relationships. While the general problem of cycle avoidance for this type of constraints is NP-complete [14], UltraBlue is a $O(DN^2)$ time heuristic algorithm (where D is the maximum constraint "fan-out" of a variable, and N is the number of constraints) that finds acyclic solution graphs while preferring constraints with higher strength. In practice, UltraBlue runs in linear time or better. UltraBlue's unique features have been designed to meet the needs of a general purpose, interactive user interface management system.

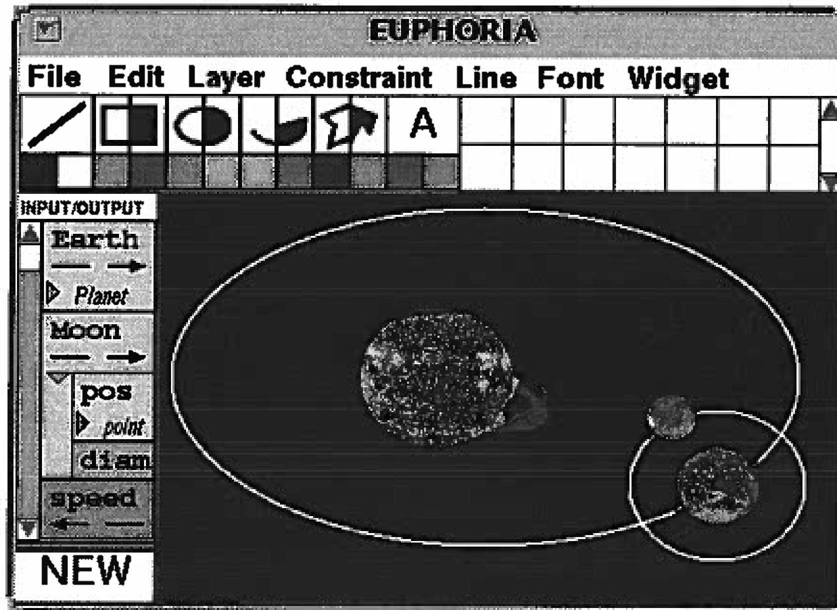


Figure 1: The EUPHORIA user interface management system running a planetary orbits simulation.

UltraBlue has been developed in the context of *The Programmers' Playground* distributed programming environment [6], [7] and the EUPHORIA user interface management system. The Programmers' Playground is a software library and run-time system for creating distributed multimedia applications. EUPHORIA is a component of the Playground system, enabling end-users to create direct manipulation, graphical user interfaces (GUIs) for distributed applications through the use of a constraint-based graphics editor (Figure 1). Features of EUPHORIA include constraint-based editing & visualization, real-time direct manipulation graphics, imaginary alignment objects, end-user definable types, end-user definable widgets with alternative representations, and support for the construction of multi-user GUIs. End-users create GUIs in EUPHORIA by simply drawing their components with a graphical editor, forming constraint relationships among the component attributes, and exposing certain attribute information to the external environment. No programming is required. The behavior of an end-user GUI is controlled by interprocess communication with external Playground modules (Section 4.3). EUPHORIA's internal architecture and communication structure utilize UltraBlue constraints in many ways, greatly reducing the amount of programming required during development. Constraints empower end-users of EUPHORIA to declaratively define graphical relationships.

This paper is organized as follows. Section 2 provides some background on constraints. Section 3 describes some of the constraint mechanisms available to EUPHORIA end-users for creating user interfaces. Section 4 discusses how constraints are used in the internal architecture of EUPHORIA. Section 5 presents the UltraBlue constraint solver algorithm. Section 6 compares the performance of UltraBlue to a comparable algorithm using three benchmark programs.

2 BACKGROUND

Much work has been done in the area of constraint maintenance. This section summaries basic terminology, the visual notation used in this paper, and related work.

2.1 Multi-way constraints

One-way dataflow constraint systems have been used extensively as a means of forming basic constraint relationships [11], [12], [18], [19]. With one-way constraints, each constraint represents a static computation with a fixed set of *input variables* (variables used by the constraint's computation) and *output variables* (variables computed by the constraint's computation). Each variable may have multiple associated constraints, forming a directed *constraint graph*. The advantage of one-way constraints is simplicity and predictability, since each constraint has a constant *computation direction* (i.e., the input and output variables are always the same). However, the static nature of one-way constraints makes their application impractical in many situations. One-way constraints force programmers to hard-code every possible computation into a constraint system, and do not allow the computation flow to change. For example, to represent the relationship among the left, right, and width of a rectangle, a "width=right-left" constraint could be created that computes the width in terms of the left and right values. However, to compute the left value it would be necessary to either manually compute it using the right and width variables (i.e., without using constraints) or add an additional constraint "left=width-right," forming a cycle of constraints.

Multi-way dataflow constraint systems allow constraints to have a dynamically changing computation flow. That is, a constraint's input variables, output variables, and its computation can vary based on the addition or deletion of constraints to the constraint graph. A set of constraints is *conflict-free* if each variable is an output variable of at most one constraint. The process of computing a solution for a set of constraints has three stages: finding a conflict-free constraint graph, forming the execution plan, and executing the plan. The first two stages are known as *planning* and the third is known as *execution*. A multi-way constraint solver redirects constraints (i.e., changing input/output direction) in order to achieve a conflict-free constraint graph. Some constraints may be left unenforced in the event of conflicting constraints. Multi-way constraints allow dynamically changing relationships to be declaratively defined, giving the constraint solver the task of determining the constraint graph's computation flow. This approach is particularly useful in interactive user interfaces, where the flow of data can change based on interaction and direct manipulation (see Section 4).

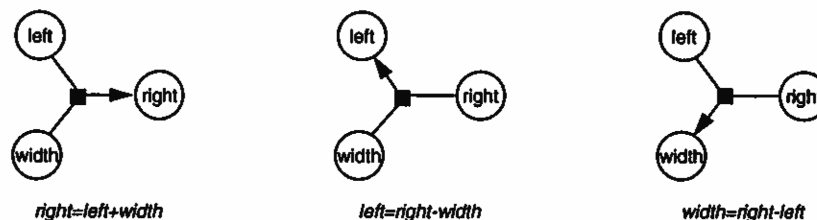


Figure 2: Possible computation directions of a multi-way width constraint.

In all figures, each variable is represented as a circle and each constraint is represented as a square with attached undirected edges to input variables and directed edges to output variables. Since this paper describes *single-output* constraints, each constraint can have only one directed edge. Figure 2 shows a multi-way constraint representing the relationship "right=left+width." This single constraint can be directed to compute any one of its associated variables using an algebraic rearrangement of the constraint's computation. That is, left can be computed with "left=right-width," and width can be computed using "width=right-left." Specifying this relationship as a multi-way constraint makes it possible to change its computation flow in the constraint graph dynamically, rather than hard-coding only one computation direction.

2.2 Constraint hierarchies and walkabout strength

While multi-way constraints simplify the task of specifying constraint relationships, their solutions can be unpredictable at times due to nondeterminism inherent in the specification. There may be many possible ways to satisfy a series of multi-way constraints. Constraint hierarchies [2], [3], [23] allow each constraint to be specified using a preference level, or *strength*, representing its relative importance. This strength information is used to determine how to enforce constraints in the event of conflicts, favoring stronger over weaker constraints. For the purposes of this paper, the following strengths (from weakest to strongest) will be used for illustration purposes: weak, medium, strong, and required.

User interface applications need efficient performance to meet the demands of real-time direct manipulation. Processing every constraint when a new constraint is added is very time consuming. Most *incremental* constraint algorithms maintain constraint information locally in a constraint graph to avoid redundant global computation. New constraints can typically be added by only considering a small subset of the constraints. UltraBlue uses the concept of *walkabout strength* [3], as a means to maintain constraint information locally at each variable. The walkabout strength of a variable represents the strength of its weakest *upstream* constraint (a constraint is said to be upstream of a variable if there exists a directed path from the constraint to the variable). When a constraint is added to a variable, this strength is used to determine whether or not to enforce the constraint and what other constraint (if any) should be unenforced to avoid a conflict. The walkabout strengths of the variables are derived from the topology of the constraint graph and its constraint strengths.

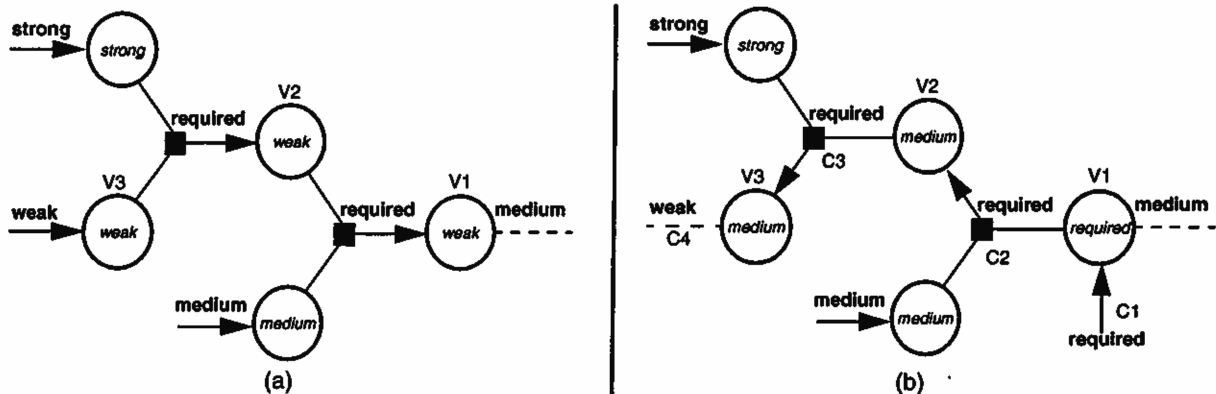


Figure 3: (a) Constraint graph with preferred constraints and walkabout strengths. (b) Adding a constraint.

For example, Figure 3a shows a constraint graph with preferential constraints and variable walkabout strengths (unenforced constraints are shown using dashed lines, constraint strengths are shown in bold, walkabout strengths are shown in italics). Variable V3 has a walkabout strength of "weak" since it is the output variable of a weak constraint. Variables V2 and V1 also have weak walkabout strengths since their weakest upstream constraint is of weak strength. Variables V1, V2, V3 form a reversed directed path to the weak constraint. Figure 3b shows the effect of adding a required constraint, C1, to variable V1. The walkabout strengths of the initial constraint graph are used to determine if C1 can be added without unenforcing more important constraints. C1 can be enforced since the walkabout strength of V1 is weaker than the strength of C1. The constraints on the reversed directed path from V1 to V3 are redirected as a result of enforcing C1, unenforcing C4 in favor of C1. The walkabout strengths of V1, V2, V3 are updated appropriately following the addition of C1.

2.3 Cycle avoidance versus cycle solving

One approach to handling constraint cycles is to use a *cycle solver.* Constraint systems supporting cycle solvers allow cyclic constraint relationships to be formed during the planning stage. During execu-

tion, each cycle's constraints are evaluated by a separate solver that attempts to solve constraint computations (e.g., through the use of a simultaneous linear equation solver). However, this approach has its disadvantages. For instance, constraint solvers typically operate on only a certain domain of computation (e.g., linear equations), limiting their application. Also, constraint solvers are not guaranteed to find a solution to an arbitrary series of equations.

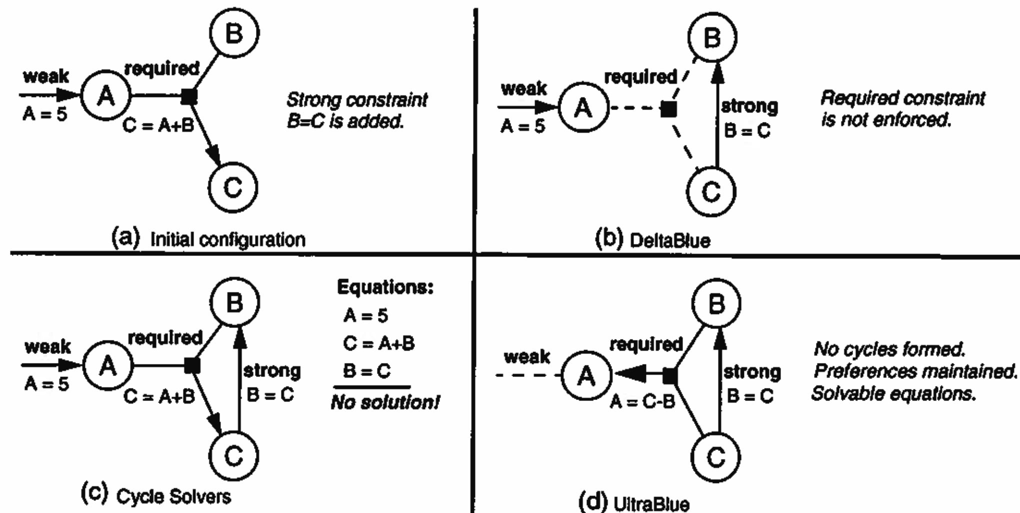


Figure 4: Example of cycle solving versus cycle avoidance constraint systems.

For example, Figure 4a shows an initial constraint graph among three variables ("A," "B," "C"), a weak constraint representing "A=5," and a required constraint representing "C=A+B." Figure 4b-d compares the effects of adding a strong constraint representing "B=C" between B and C using the DeltaBlue algorithm, a cycle solving algorithm, and the UltraBlue algorithm. The new constraint forms a cycle in the constraint graph; each algorithm has a different approach in handling the cycle. DeltaBlue unenforces an arbitrary constraint on the cycle (Figure 4b). In this case, the required constraint is unenforced while the weak and strong constraints remain enforced. This is undesirable since the required constraint is the most important constraint, and should be enforced. An algorithm that uses a cycle solver would allow a cycle to be formed between B and C during the planning stage (Figure 4c). However, the solver would fail during the execution stage since the cycle's equations do not have a solution. The UltraBlue algorithm redirects the required constraint so that a cycle is avoided (Figure 4d). The weak constraint is unenforced in favor of the required and strong constraints, providing a solvable series of equations.

We have found that the formation of cyclic constraint relationships is quite common, especially when end-users are given the ability to form arbitrary constraints (Section 3). Unsolvable cycles similar to Figure 4c are also commonly formed in interactive applications. UltraBlue was developed as a heuristic approach to eliminate cycles in multi-way constraint graphs while preferring constraints of higher strength.

2.4 Related work

This section summarizes some related work in the area of constraints. A discussion of related work in the area of user interfaces can be found elsewhere [17].

The SketchPad system [24] was the first graphical system to use constraints. SketchPad allowed users to assert relationships between graphical objects and showed the effects of the constraints during real-time direct manipulation. To handle cyclic constraint relationships, SketchPad employed relaxation, an iterative error minimizing process to execute constraints. However, relaxation tends to be slow to

converge and sometimes gets stuck in a local minimum. The Bramble toolkit [4] provided support for graphical manipulation by employing differential constraint techniques. A constraint engine capable of managing non-linear equations is used to map interactive controls and constraints to graphics object parameters. However, this approach is only applicable to time-based, continuous motion interaction rather than general purpose computation.

Many one-way constraint solvers use a "once around the loop" approach in handling cycles. That is, cycles are permitted to be formed, and the computation of each constraint on a cycle is executed once during constraint evaluation. Systems that use this approach include Fabrik [13], RENDEZVOUS [10], [11], and Hudson's incremental attribute evaluation algorithm [12]. However, this method of evaluation does not avoid unsolvable series of equations (see Section 2.3), and can leave constraint computations unsatisfied.

The DeltaBlue algorithm [3], [14], [23] provides support to solve hierarchies of multi-way, single-output dataflow constraints. DeltaBlue uses a comparator known as "locally-predicate-better" to decide which constraints should be enforced. Locally-predicate-better is a metric that prefers to enforce stronger constraints over (possibly many) weaker constraints [3]. DeltaBlue does not attempt to avoid cycles of constraints. When DeltaBlue detects a constraint cycle, it arbitrarily unenforces one of the constraints on the cycle, regardless of its strength (see Section 2.3). In contrast, our algorithm, UltraBlue, uses a heuristic that attempts to maintain locally-predicate-better condition even in the case of cycles. However, the general problem is NP-complete [14], so when cycles are eliminated UltraBlue may not find an optimal solution according to the locally-predicate-better comparator.

Typically, dataflow constraints are used to compute equation-based, equality relationships among a set of variables. In this case, the value of a variable is determined by possibly many upstream variable values in the constraint graph. DeltaBlue, as well as other dataflow algorithms, cannot manage inequalities and other assertions in a way that is consistent with the multi-way ability of constraints. While a constraint method is free to limit the range of its output variable values, this often results in upstream variable values which are inconsistent with downstream variable values. Unlike DeltaBlue, UltraBlue supports a value consistency mechanism that allows inequalities and other assertions to be formed on multi-way constrained values.

As discussed earlier, some constraint systems allow cyclic relationships to be formed in the planning stage and use a cycle solver (e.g., using a linear equation solver). This approach can generate a series of constraints whose computations are not solvable (see Section 2.3). The Magritte system [8] used constraints in an editor for simple line drawings. Algebraic transformations were used to eliminate cycles in the constraint graph. When propagation encountered a cycle, the transformation system attempted to replace each cycle with a single complex constraint (i.e., executed through the use of a cycle solver). The SkyBlue algorithm [21] provides support for solving hierarchies of multi-way, multi-output constraints, and has been applied in the Garnet toolkit [22]. Multi-output constraints provide a convenient way to group multiple related computations into a single constraint. SkyBlue allows cyclic relationships to be formed during the planning stage. During execution, each cycle is essentially treated as a single multi-output constraint that can be executed using a cycle solver. However, SkyBlue has an exponential running time (the problem of maintaining this type of constraints is NP-complete). The QuickPlan algorithm [25] can solve any hierarchy of multi-way, multi-output, dataflow constraints but adds some restrictions to the general problem. QuickPlan is guaranteed to solve a series of constraints in polynomial time provided that there exists at least one acyclic, conflict-free solution. QuickPlan does not attempt to avoid cyclic constraint relationships.

3 END-USER CONSTRAINT INTERACTION IN EUPHORIA

UltraBlue has been used extensively in the EUPHORIA user interface management system for more

than a year. EUPHORIA has been used in an undergraduate course at Washington University focusing on the development of distributed multimedia applications. Washington University's Laboratory for Pen Computing and Visual Programming also has plans to apply UltraBlue to the Picasso graphics editor. This section describes some of the ways that EUPHORIA end-users can utilize constraints in the construction of direct manipulation GUIs.

3.1 End-user defined constraints

EUPHORIA provides an editor (see Figure 1) that enables end-users to graphically define persistent relationships among graphics objects. As in most graphics editors, each graphics object has a number of *handles* that may be used to manipulate graphics object attributes (e.g., resizing a shape). In addition, handles may also be used as data ports for establishing constraint relationships. An *equality constraint* is established between two graphics object attributes by drawing a connection line between the corresponding graphics object handles. This approach is based on a visual language with consistent visual semantics for defining both interprocess and intraprocess communication [16].

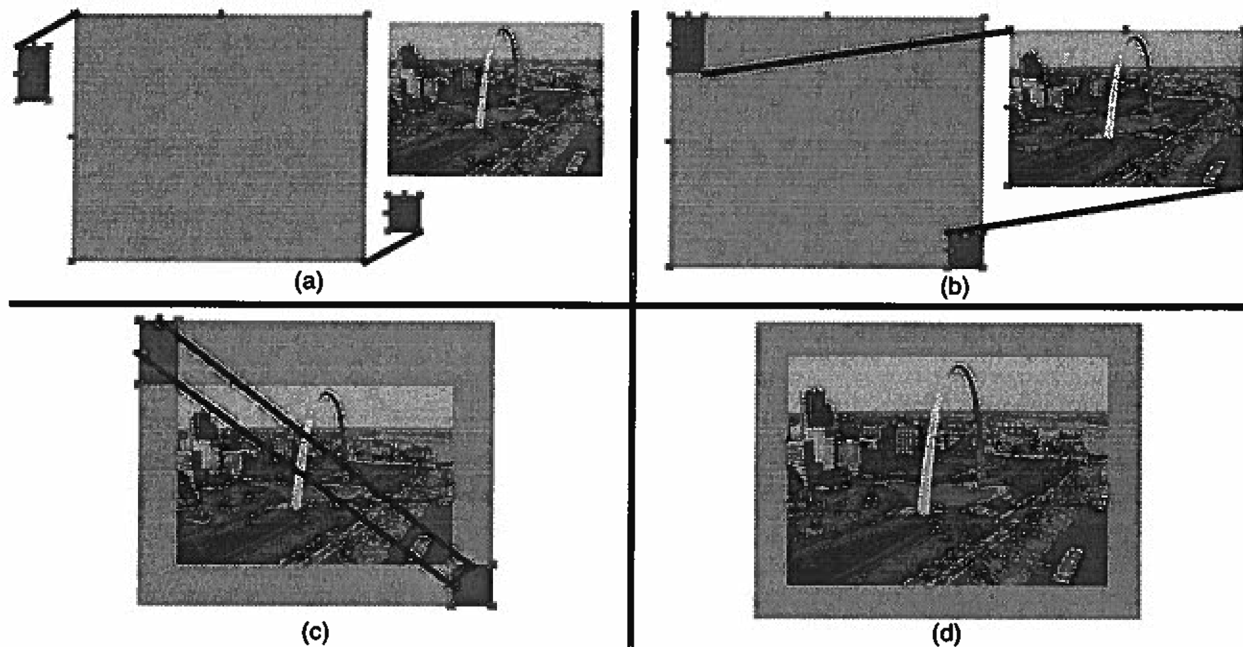


Figure 5: End-user creation of an adjustable picture frame using constraints.

For example, suppose that an end-user needed to construct an adjustable picture frame for a photograph. That is, the photograph should be surrounded by a border of uniform thickness and both the photograph and border should be adjustable through direct manipulation. Figure 5 illustrates how this frame may be constructed by an end-user through the use of constraints. First, three rectangles representing the frame and border offsets are drawn. The corners of the offsets are attached to the frame by drawing constraint lines among the appropriate graphics object handles (Figure 5a). The offsets are instantaneously snapped into place when each constraint is established. Each rectangle may still be independently adjusted, except that the offsets always remain attached to the opposite corners of the frame. Second, the picture is circumscribed into the frame by establishing constraints among the inner corners of the offset rectangles (Figure 5b). This action snaps the picture into place while still allowing the admissions of both the picture and the frame to be manipulated. Third, the size of the offset rectangles are constrained to be equal by connecting their width and height handles (Figure 5c). Also, the lower offset rectangle is set to be a square by constraining its width and height to be equal. Finally, the offset rectangles are hidden, resulting in an adjustable frame surrounding the picture (Figure 5d). Note that this

example involves constraint cycle avoidance since the added constraints cause potential cycles with the rectangles' internal constraints (see Section 4.1).

In addition to equality constraints, EUPHORIA also supports constant and conversion constraints. A *constant constraint* is formed on a graphics object attribute by "anchoring" its corresponding handle. A *conversion constraint* is a specialized equality constraint in which the graphics attribute data types are compatible, but not the same, requiring a type conversion (e.g., connecting a real number graphics attribute to a string graphics attribute).

3.2 End-user constraint strength specification

The strength of a constraint may be specified by an end-user. Constraint strengths are used to establish behavioral preferences among graphical constraint relationships. In the event that constraints conflict or there are cycles, the strengths are used to resolve the conflicts and cycles, favoring stronger over weaker constraints. Manipulation of graphics objects is also achieved through constraints (see Section 4.2). The strength of various types of manipulation actions may be adjusted, modifying the manipulation behavior.

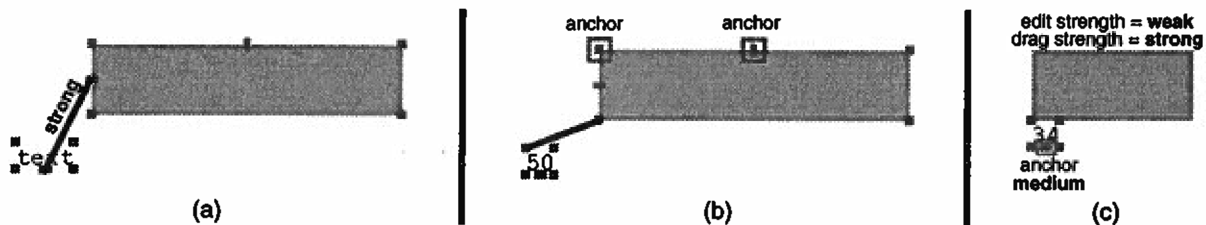


Figure 6: End-user creation of an interactive bar graph with a digital read-out.

For example, Figure 6 shows the process of creating an interactive bar graph with an associated digital read-out. First, a rectangle and text object are created (Figure 6a). A strong conversion constraint is formed between the rectangle's width handle and the text object's string handle. As a result, the text displays the value of the rectangle's width, even during direct manipulation of the rectangle. Second, the text is made adjacent to the rectangle's bottom through an equality constraint between the positions of the rectangle and the text; the position and height of the rectangle are anchored through the use of constant constraints (Figure 6b). At this point, the text may also be edited, changing the width of the rectangle during user typing. However, since the text is only intended to serve as a digital read-out (not an editable label) in this application, the string attribute of the text is then anchored with a medium strength constant constraint (Figure 6c). This disables direct editing of the text since the constant constraint's strength (medium) is stronger than the edit text strength manipulation action (weak). However, direct manipulation of the bar graph still results in the textual display of the bar's width since the conversion constraint strength (strong) and the manipulation drag strength (strong) override the anchor's strength (medium).

3.3 Constraint visualization & editing

EUPHORIA supports visualization and editing of constraints, allowing end-users to optionally view constraints of selected graphics objects, view current computation directions, delete constraints, and change constraint strengths. A constant constraint is visually represented as a rectangle surrounding the constraint's corresponding handle. An equality or conversion constraint is visually represented as a directed arrow between graphics object handles. If there is not sufficient space to display a directed arrow for an equality or conversion constraint, then the constraint is visually represented as a circle. All types of constraints may be deleted or modified through interaction with the constraint visualization.

Constraint computation directions are helpful in understanding how objects of a constrained drawing interact. End-users can see how constraints are enforced, as well as which constraints cannot be enforced

(unenforced constraints are represented as undirected, dashed lines). However, crowded drawings cannot be effectively visualized in this way since multiple handles and constraints may be in close proximity. For this reason, a *taffy pull mode* was developed to enable users to temporarily pull apart constrained graphics objects while viewing their constraint relationships. Graphics objects can be moved apart freely without viewing the effects of the constraints. The constraint visualizations are "stretched," allowing users to see more information.

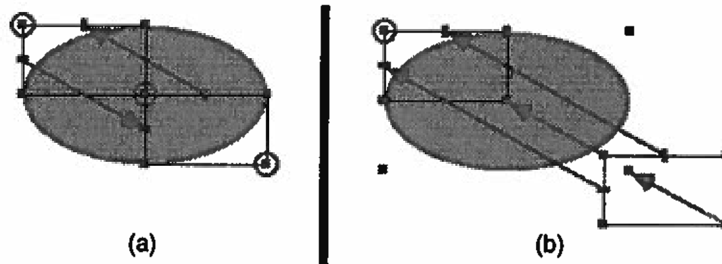


Figure 7: Constraint visualization in (a) normal view and (b) taffy pull mode.

An oval in EUPHORIA does not have a handle in its center. However, a center handle may be established by constraining two rectangles within an oval and constraining the sizes of the rectangles to be equal. Figure 7a shows how these graphics objects appear when constraint visualization is enabled. The constraints between the point graphics attributes are represented as circles around the appropriate handles. Figure 7b shows the graphics objects appear when viewed in taffy pull mode. The "apparent position" of the lower rectangle is displaced temporarily by dragging the rectangle, allowing the equality constraints between the center handles and the bottom right handles to be viewed and edited.

4 CONSTRAINT APPLICATIONS IN THE EUPHORIA ARCHITECTURE

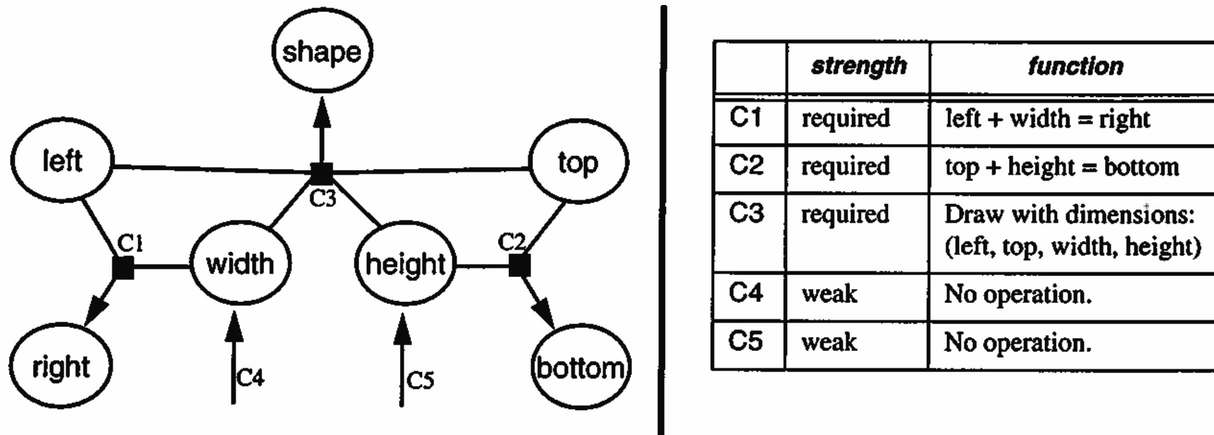
In addition to maintaining end-user defined relationships, constraints are also used by EUPHORIA to represent the internal relationships among window components (i.e., window layout), graphics objects, and graphics object handles. UltraBlue's value consistency and cycle avoidance features are needed to support this functionality. Through this uniform constraint mechanism, the implementation of EUPHORIA is greatly simplified. This section describes some of these constraint applications in detail.

4.1 Constraint representation of graphics objects

Internally, the relationships among the attributes of a graphics object are represented as a constraint graph. Figure 8 shows the constraint graph for a rectangular bounded shape (e.g., the rectangles and image from Figure 5), with variables for attributes such as left, right, width, etc. Constraints and their associated strengths are used to maintain relationships and the behavior of the shape. Constraints C1 and C2 compute the sum function among the shape's dimension variables. These multi-way constraints allow any one of their associated variables to be computed in terms of the others. The strength of these constraints is "required" since they should always be maintained in the presence of conflicting constraints. In addition, the width and height variables have value consistency assertions that prevent their values from becoming negative.

Constraint C3 is known as an *active value* constraint [9]. The purpose of an active value constraint is to perform a "side effect" action whenever it is executed. In this case, C3 is used to draw the rectangular shape whenever one of its dimension variables is modified[1]. Since C3 represents a side-effect action, and not an equation, it is represented as a one-way constraint with "shape" as its only output variable. This variable contains an implementation dependent representation (e.g., a pointer to a rectangle class with

[1]The shape is invalidated for later redrawing [17].



	strength	function
C1	required	left + width = right
C2	required	top + height = bottom
C3	required	Draw with dimensions: (left, top, width, height)
C4	weak	No operation.
C5	weak	No operation.

Figure 8: Constraint graph for a rectangular bounded shape.

associated X Windows information) that is used in drawing the shape.

Constraints C4 and C5 are known as *stay* constraints. The purpose of a stay constraint is to state the preference that a variable should remain constant in the absence of stronger, conflicting constraints. Stay constraints are also used to specify the behavior of a constraint graph when constraint relationships are underconstrained. As with other constraints, the strength of a stay constraint represents its importance. In this case, C4 and C5 are of strength "weak," keeping the width and height variables constant in the absence of stronger conflicting constraints. The stay constraints on width and height keep the size of a shape constant as it is moved.

As mentioned earlier, shapes are manipulated through the use of constraints. Whenever a rectangular shape is moved, *edit* constraints (not shown) are added to some of its variables. A variable with an enforced edit constraint is not computed in terms of any other constraint or variable, allowing the system to safely set its value externally. In this case, as the shape is dragged by the user, its associated position is placed into the left and top variables. These values are propagated through the constraint graph during execution, resulting in constraint computations and active value side-effects. That is, the values of related quantities such as the "right" and "bottom" attributes are computed, and the connected shape(s) are redrawn. In this way, complex series of connected shapes may be manipulated in real-time without need for the intermediate "xor outline" approach that most graphics editors and window systems use.

4.2 Constraint representation of graphics object handles

Constraint graphs of a graphics object can be connected directly to constraint graphs of their associated handles. In this way, one can manipulate a graphics object using the same mechanism that is used for other types of internal communication. Figure 9a shows the constraint graph representation of a rectangle with an attached handle on its left-top corner. Attaching a handle simply involves creating a constraint graph for the handle and forming constraints between the handle's graph and the rectangle's graph (see Figure 9a, constraints C1 and C2). In this way, whenever the rectangle or the handle change position or size in any way (e.g., direct manipulation, external interprocess communication, etc.) the other is changed and redrawn to be consistent with that change.

To resize the rectangle, a few other constraints are added to the graph. In Figure 9b, the right and bottom constraint variables are anchored with strong constant constraints (C3 and C4). The left and top handle constraint variables are set editable by attaching medium edit constraints (C5 and C6). The strengths of these constraints are stronger than that of the stay constraints on the size of the rectangle (C7 and C8, weak strength). The stay constraints are unenforced in favor of the edit constraints, resulting in a redirected constraint graph that computes the rectangle's size in terms of the handle's position. When-

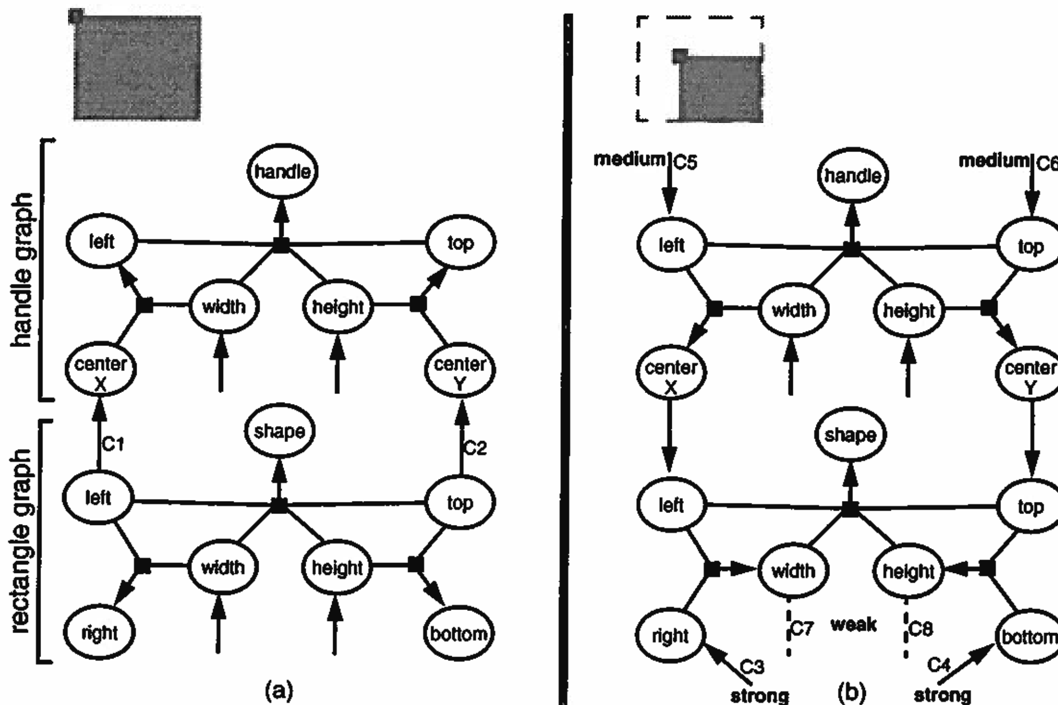


Figure 9: (a) Attaching a graphics object handle to a rectangle's constraint graph. (b) Dragging the handle.

ever the handle is moved by the user, the x and y coordinates of the handle are copied into the handle's left and top constraint variables. Execution of the constraint graph results in: (1) redrawing the handle, (2) computing a new size for the rectangle, (3) redrawing the rectangle, and (4) propagation of values to any other objects that may be attached to the graph.

Each other graphics object handle is added to the rectangle's constraint graph in a similar way. No special purpose programming was required to create each individual handle. Other handles use an identical constraint graph structure, but are connected to different variables of the rectangle's constraint graph (e.g., a right-bottom handle is created by attaching the handle's constraint graph to the right and bottom variables rather than the left and top variables shown in Figure 9).

Similarly, this constraint graph approach is augmented to support a number of other features such as constraint visualization and type conversion. The taffy pull mode described earlier was implemented by adding additional "offset" variables to the rectangle's constraint graph, in order to distinguish between the real position of a rectangle and its apparent position. Active value constraints are used to visualize end-user defined constraints.

4.3 Interprocess communication & constraints

In Playground, a distributed application consists of a number of *modules* (i.e., processes), each having a set of externally readable and/or writable *published* data structures. EUPHORIA is one such module, allowing state information of a user interface to be published for use in external applications (shown as color-coded squares in Figure 10a). Interprocess communication is specified graphically at run-time by creating *logical connections* among the modules' published data structures. Communication among modules occurs implicitly; when a published data structure is modified, the new value is automatically sent to its connected data structures. In this way, a user interface is completely decoupled from its application.

For example, an interactive bouncing ball could be created as follows. An oval representing the ball is drawn in EUPHORIA. The oval is set to be a circle by constraining its width and height to be equal.

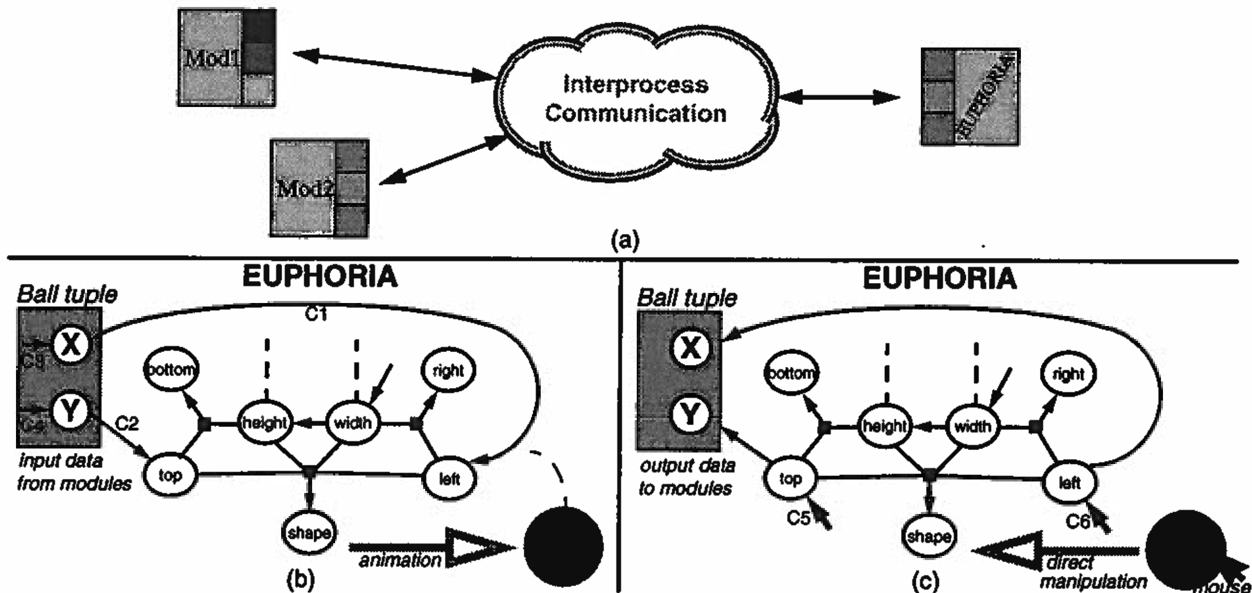


Figure 10: (a) Module interprocess communication. (b) Animation of a bouncing ball. (c) Direct manipulation of a ball.

The width is then anchored with a constant constraint in order to make the size constant. Position information about the ball is exposed to external modules by publishing the ball's left-top coordinate attribute (i.e., a tuple of x, y values). This exposed state is connected to a general-purpose physics module that simulates the effects of gravity over time as well as reacting to user interaction.

When external values are communicated to EUPHORIA's published data structures, EUPHORIA reacts by copying the values into their associated constraint variables. These constraint variables are connected to the constraint graphs of graphics objects within EUPHORIA. The values of these variables are propagated during constraint evaluation, having the effect of changing graphics object attributes and redrawing the objects according to the established constraint relationships. In the same way, end-user changes to graphics objects (e.g., direct manipulation) are propagated through the constraint graph and copied into EUPHORIA's published variables, sending new values out to external applications.

For example, Figure 10b shows a bouncing ball, its constraint graph, and its published position tuple. The fields of the published tuple are connected to the ball's constraint graph through the use of equality constraints (C1 and C2). To send values into the constraint graph, edit constraints are formed on the x, y variables associated with the tuple (C3 and C4). Values received from external modules copied into the x and y variables and are propagated through the constraint graph. The result is an animated ball under the control of an external module. However, the user may also drag and throw the ball through direct manipulation (Figure 10c). When the ball is grabbed by the mouse, edit constraints are formed on the left and top variables of the ball's constraint graph (C5 and C6), changing the computation direction of the constraint graph. During user manipulation, the mouse coordinates are copied into the ball's left and top variables, propagating the values out to the published ball tuple. The external physics module reacts to these interaction values, changing the direction and magnitude of the ball appropriately.

5 ULTRABLUE MULTI-WAY CONSTRAINT ALGORITHM

This section describes the UltraBlue constraint solver algorithm. A C++ version of UltraBlue has been used in the implementation of EUPHORIA. In viewing the pseudocode of this section, the reader should note that no global variables are utilized; each variable in a function is either a function parameter or a locally defined variable. Also, all parameters are passed by reference and all function return values are references to data (i.e., no copying of data structures is assumed).

Section 5.1 - Section 5.3 describes the required portion of the UltraBlue algorithm for maintaining an acyclic graph of hierarchies of multi-way constraints. Section 5.4 describes an efficient algorithm for evaluating a series of constraints in topological order. Section 5.5 presents two optional extensions to UltraBlue for limiting the computation direction of constraints and value consistency enforcement.

5.1 Strength

The **strength** of a constraint represents its relative preference level in relation to other constraints. The "walkabout" strength of a variable represents its derived strength based on the strengths of all of its upstream constraints. When constraints conflict, or cycles need to be resolved, variable walkabout strengths are used to determine how to solve the series of constraints. Two strengths, *weakest* and *strongest*, are reserved. There can be any number of constraint levels, but a user constraint must be stronger than *weakest* and weaker than *strongest*.

```
Weaker(s1, s2: Strength) : Boolean
(1) return true iff s1 is weaker than s2
```

5.2 Variable

Table 1 describes the variable structure, consisting of a number of fields for maintaining variable values, constraint graph topology, and other internal bookkeeping information.

field	*type*	*initial value*	*description*
value	any type	user supplied	current value
constraints	Set of Constraints	∅	constraints that reference this variable
determinedBy	Constraint	`null`	constraint that computes this variable, or `null`
walkStrength	Strength	**weakest**	walkabout strength
count	Integer	0	used for topological ordering
mark	Symbol	a unique mark	used for cycle detection
breakPoint	Constraint	n/a	used to find cycle breakpoints

Table 1: Variable fields.

```
PropagateStrength(v: Variable)
(1)  for each c ∈ ConsumingConstraints(v), str ← WalkStrength(c)
(2)      such that str ≠ c.output.walkStrength do
(3)          c.output.walkStrength ← str
(4)          PropagateStrength(c.output)
```

`PropagateStrength` computes the walkabout strength of all variables downstream from a given variable. Weaker strengths are propagated throughout the constraint graph, forming reversed directed paths to unconstrained variables and constraints of lower strength.

```
ConsumingConstraints(v: Variable) : Set of Constraints
(1)  return {c ∈ v.constraints | Enforced(c), c.determinedBy ≠ v}
```

`ConsumingConstraints` returns a variable's subset of constraints that are currently using the variable as an input to their computation.

5.3 Constraint

Table 2 describes the constraint structure, consisting of a number of fields for maintaining constraint

graph topology, computation, and other internal bookkeeping information. The `variables` field is of type "VariableSeq," an ordered list of variables. This list and the `output` field are passed as parameters to the `method` function when the constraint is executed.

field	***type***	***initial value***	***description***
variables	VariableSeq	user supplied	variables referenced by this constraint
output	Variable	**null**	output variable of this constraint, or **null**
method	Method	user supplied	function to compute the value of an output variable
strength	Strength	user supplied	preference level in the constraint hierarchy

Table 2: Constraint fields.

```
Enforced(c: Constraint) : Boolean
(1)  return (c.output ≠ null)
```

Enforced returns true if and only if a supplied constraint computes an output variable.

```
WalkStrength(c: Constraint) : Strength
(1)  return weakest strength of c and
(2)         the walkStrengths of {v ∈ c.variables | v ≠ c.output}
```

`WalkStrength` is used to compute a variable's walkabout strength from its computing constraint (unconstrained variables are of *weakest* strength). This strength represents the weakest strength of an upstream constraint or unconstrained variable.

5.3.1 Adding & removing a constraint

```
AddConstraint(c: Constraint)
(1)  for each v ∈ vs do
(2)     v.constraints ← v.constraints ∪ {c}
(3)  Enforce(c)
```

`AddConstraint` connects the constraint to each of its associated variables and attempts to enforce the constraint. The `Enforce` function is then called to determine the propagation directions of the connected constraints and the walkabout strengths of connected variables.

```
RemoveConstraint(c: Constraint)
(1)  for each v ∈ c.variables do
(2)     v.constraints ← v.constraints - {c}
(3)  if Enforced(c) then
(4)     out ← c.output
(5)     Unenforce(c)
(6)     for each c1 such that c1 is downstream from out, ¬Enforced(c1) do
(7)        Enforce(c1)
```

`RemoveConstraint` disconnects the constraint from each of its associated variables. If the constraint was enforced previously, it attempts to enforce each of its downstream constraints that are not currently enforced.

```
Unenforce(c: Constraint)
(1)  out ← c.output
(2)  c.output ← null
(3)  out.determinedBy ← null
(4)  out.walkStrength ← weakest
(5)  PropagateStrength(out)
```

The Unenforce function unenforces a constraint so that it does not compute any variable. Its previous output variable is updated to be unconstrained.

5.3.2 Enforcing a constraint

```
Enforce(c: Constraint)
(1)  if ¬Enforced(c) then
(2)     updateVars ← Ø
(3)     RedirectPath(c, null, updateVars)
(4)     if Enforced(c) then
(5)        updateVars ← updateVars ∪ c.variables
(6)        Execute the methods of updateVars' downstream constraints
```

A constraint is enforced by redirecting the propagation direction of a series of constraints. This redirection may involve resolving cycles that are formed as a result of the constraint redirection. When the redirection of the constraint graph is complete, the methods of each affected constraint are executed, computing the values of the affected variables.

```
RedirectPath(c: Constraint; v: Variable; updateVars: Set of Variables)
(1)  path ← Ø
(2)  retracted ← c
(3)  repeat
(4)     v ← SelectOutput(retracted, v)
(5)     if v ≠ null then
(6)        path ← path ∪ {retracted}
(7)        retracted.output ← v
(8)        newRetracted ← v.determinedBy
(9)        v.determinedBy ← retracted
(10)       if newRetracted ≠ null then
(11)          retracted ← newRetracted
(12)          retracted.output ← null
(13) until v = null or newRetracted = null
(14) if Enforced(c) then
(15)    c.output.walkStrength ← WalkStrength(c.output.determinedBy)
(16)    PropagateStrength(c.output)
(17)    ResolveCycles(path, updateVars)
```

RedirectPath attempts to enforce a constraint by redirecting a path of connected constraints. A constraint can be enforced if there is either an unconstrained variable or a weaker constraint upstream. The walkabout strength of each connected variable is used to locally determine if one of these conditions are satisfied. The loop in lines 3 - 13 redirects a path of constraints until an unconstrained variable or weaker upstream constraint is encountered. The weaker upstream constraint, if any, is left unenforced. If the new constraint is successfully enforced, then the walkabout strengths of all downstream variables are updated and the cycles (if any) originating from the path's constraints are resolved.

```
SelectOutput(c: Constraint; prev: Variable) : Variable
(1)  candidates ←
(2)     {v ∈ c.variables | v ≠ prev, Weaker(v.walkStrength, c.strength)}
(3)  if candidates ≠ Ø then
(4)     return v ∈ candidates such that v.walkStrength is the weakest
(5)  else return null
```

SelectOutput selects the output variable of a constraint from its set of variables. The selected output is the variable with the weakest walkabout strength that is also weaker than the strength of the constraint. This selection determines how to redirect a constraint during the RedirectPath function.

5.3.3 Breaking cycles

UltraBlue employs a heuristic algorithm for avoiding cyclic constraint relationships while attempting to satisfy the "locally-predicate-better" comparator (see Section 2.4). The heuristic involves eliminating cycles by redirecting the computation flow of one or more constraints on a cycle. To make this task tractable, the source constraint (i.e., the constraint that caused the cycle) is temporarily disconnected from its input variables. In situations where many cycles are resolved simultaneously (e.g., if redirecting a constraint on a cycle causes other cycles), the source constraints limit the amount of processing done in finding an acyclic graph since constraints become disconnected. However, this may cause stronger constraints to be left unenforced (i.e., non-optimal solution graphs). In general, this results from highly constrained systems that cannot be efficiently solved.

```
ResolveCycles(path: Set of Constraints; updateVars: Set of Variables)
(1)  sources ← an empty ConstraintSeq
(2)  oldVariables ← a mapping from constraints to variables
(3)  check for cycles from path
(4)  for each c ∈ path such that c is on a cycle then
(5)        oldVariables[c] ← c.output
(6)        Insert c into sources, sorted by decreasing strength
(7)        c.output.determinedBy ← null
(8)        c.output ← null
(9)  for each c ∈ sources in order, v ← oldVariables[c]
(10)    such that ¬Enforced(c), v.determinedBy = null do
(11)       c.output ← v
(12)       c.output.determinedBy ← c
(13)       EliminateCycle(c, updateVars)
```

Function `ResolveCycles` is called to resolve the cycles that may have been created by redirecting a path of constraints (passed as the `path` parameter). Each cycle is eliminated by redirecting and/or unenforcing constraints. Since it was assumed that there were no cycles before `RedirectPath` was called, each of the current cycles is the result of a redirected constraint in `path` or the newly added constraint. The loop in lines 4 - 8 temporarily breaks all of these cycles to facilitating further processing. That is, each constraint in `path` that has a cycle is locally unenforced so that the cycles can be processed independently. The loop in lines 9 - 13 re-enforces each constraint independently, eliminating its cycles separately. The constraints are re-enforced in order of strongest to weakest favoring stronger constraints over weaker constraints. An alternative way to implement the above "oldVariables" mapping functionality is to add an "oldVariable" field to `Constraint`, assigning and accessing the oldVariables field in lines 5 and 9.

```
EliminateCycle(c: Constraint; updateVars: Set of Variables)
(1)  Disconnect c from each of its variables, except c.output
(2)  v ← c.output
(3)  v.walkStrength ← strongest
(4)  PropagateStrength(v)
(5)  repeat
(6)     Create unique marks "mark" and "cmark"; mark c's inputs with cmark
(7)     b ← FindBreak(c, mark, cmark)
(8)     if b ≠ null and b ≠ c then
(9)        old ← b.output
(10)       Unenforce(b)
(11)       RedirectPath(b, old, updateVars)
(12) until b = null or b = c
(13) Reconnect c to its variables
(14) if b = c then
(15)    old ← c.output
(16)    c.output ← null
(17)    old.determinedBy ← null
(18)    RedirectPath(c, old, updateVars)
(19)    updateVars ←updateVars ∪ {old}
(20) if v.determinedBy = null then
(21)    v.walkStrength ← weakest
(22) else v.walkStrength ← WalkStrength(v.determinedBy)
(23) PropagateStrength(v)
```

Function `EliminateCycle` eliminates all cycles associated with a supplied constraint, `c`. The cycles are eliminated by redirecting at least one constraint on each of `c`'s cycle(s). It is assumed that when this function is called, the only cycles that exist in the constraint graph were formed as a result of changing the propagation direction of `c`. Lines 1 - 4 temporarily eliminate all of the cycles by disconnecting `c` from its input variables, simplifying the cycle resolution process. The output variable of `c` is given a temporary *strongest* strength, in order to ensure that it will not be redirected in lines 5 - 19.

The loop in lines 5 - 12 eliminates each of the former cycles associated with `c` (i.e., paths from `c`'s output to one of `c`'s inputs). During each iteration, a cycle "breakpoint" is computed, representing the best constraint on `c`'s associated cycle(s) to be redirected or unenforced. Breakpoint constraints are redirected, eliminating these cycles. The loop terminates when there are no more cycles or `c` is determined to be the best breakpoint; `c` is then reconnected to its input variables. If the breakpoint is `c`, it is redirected so that it is not on the current cycle. The walkabout strengths of all downstream variables are recomputed, overriding the earlier strongest strength.

```
FindBreak(c: Constraint; mark, cycleMark: Symbol) : Constraint
(1)  if c.output.mark = cycleMark then
(2)     return c.output.determinedBy
(3)  if c.output.mark = mark then
(4)     return c.output.breakPoint
(5)  best ← null
(6)  for each c1 ∈ ConsumingConstraints(c.output) do
(7)     back ← FindBreak(c1, mark, cycleMark)
(8)     if back ≠ null and
(9)      (best = null or Weaker(WalkStrength(best), WalkStrength(back))) then
(10)       best ← back
(11) c.output.breakPoint ← best
(12) c.output.mark ← mark
(13) if best = null or Weaker(WalkStrength(best), WalkStrength(c)) then
(14)    return best
(15) else return c
```

FindBreak returns the best cycle breakpoint, if any, associated with a supplied constraint, c. If there is only one cycle, FindBreak returns the constraint on the cycle having the weakest input walkabout strength (i.e., the walkabout strength of c's weakest input variable). If there are multiple cycles associated with c, the strongest cycle breakpoint is returned. That is, for each cycle associated with c, there is a breakpoint having the weakest input walkabout strength; of all such breakpoints, the breakpoint having the strongest input walkabout strength is returned. The cycles of the stronger breakpoints are resolved first so that relatively weak constraints can be overridden in favor of stronger constraints.

5.3.4 Time complexity

The following analysis determines the time complexity of adding a constraint to a constraint graph. Let *N* represent the number of constraints in the graph. Let *D* represent the maximum degree of a variable (i.e., the constraint "fan-out" of a variable). It is assumed that the degree of a constraint is bounded by a fixed constant; in practice, this value is rarely greater than five or six. In practice, D also tends to be small in comparison to N. However, one could construct a constraint graph with D as a function of N (see the Star Benchmark, Section 6.1).

The time complexity of adding a constraint, assuming that *no cycles are formed*, is O(N). The loop in lines 3-13 of function RedirectPath can potentially redirect every constraint. Strength propagation in PropagateStrength, occurring after all constraints are redirected, is O(N) since each constraint is visited at most once. Redundant propagation is eliminated through the use of a conditional on the propagation strength. WalkStrength operates in constant time since the degree of a constraint is bounded by a fixed constant. Cycle resolution in ResolveCycles is also O(N), since it may require a search of the entire graph for cycles (marking of variables can be used to avoid redundant search).

The time complexity of adding a constraint, when *cycles are formed and resolved*, is $O(DN^2)$. Cycle resolution is called after constraint redirection occurs (see above). RedirectPath calls ResolveCycles, which calls EliminateCycle, calling FindBreak and recursively calling RedirectPath. FindBreak runs in O(DN) time since each N constraint is traversed at most once (due to variable marking); for every constraint traversed, each of its output variable's D consuming constraints is processed in the loop of lines 7 - 10. The time to execute EliminateCycle (not including recursive calls to RedirectPath) is bounded by O(DN) from FindBreak. Since a constraint is disconnected from its input variables in line 1 of EliminateCycle, EliminateCycle can be recursively called at most N times. This restricts both the number of iterations of the loop in lines 5 - 12 of EliminateCycle and the loop in lines 8 - 12 of ResolveCycles to a total of N iterations. The loop in lines 4 - 8 of ResolveCycles takes $O(N^2)$ since there may be N constraints in the path, and each iteration of insertion sort of line 6 takes O(N) time. The running time of ResolveCycles (and, hence, AddConstraint) is $O(DN^2)$ since the call to EliminateCycle in line 12 can be invoked recursively at most N times, executing in O(DN) time.

In practice, the running times of the above operations are much better than their worst case time complexities. Due to the incremental nature of the algorithm, most operations are performed in linear time or better since only a small portion of the constraint graph may be modified at any given time (see the Tree Benchmark, Section 6.2).

5.4 Plan

Many times, the values in a constraint graph are updated repeatedly based on a fixed subset of changing variable values. For example, Section 4.2 describes the use of constraints to implement a graphics object handle; dragging a handle involves a continuous update of constraint variable values. It is helpful to save the ordering of constraint evaluation as a *plan* for efficient updates. Also, the order of constraint evaluation is critical; a plan that updates constraints more than once can seriously impact the performance of a graphical application. This section describes an efficient algorithm for creating and

evaluating a topologically ordered plan based on a set of changing variable values. Section 5.5.2 discusses an extension to the plan algorithm, providing a general purpose value consistency mechanism for maintaining arbitrary assertions.

```
CreatePlan(inputs: Set of Variables) : ConstraintSeq
(1)  for each v ∈ inputs do
(2)     SetCount(v)
(3)  cs ← an empty ConstraintSeq
(4)  for each v ∈ inputs do
(5)     FormPlan(cs, v)
(6)  return cs
```

`CreatePlan` takes as a parameter a set of variables whose values are to be propagated through the constraint graph (known as "`input`" variables of the plan). An ordered list of constraints is created and returned by topologically traversing the constraint graph from the supplied variables to all dependent, downstream constraints.

```
SetCount(v: Variable)
(1)  for each c ∈ ConsumingConstraints(v), out ← c.output do
(2)     out.count ← out.count + 1
(3)     if out.count = 1 then
(4)        SetCount(out)
```

`SetCount` traverses the constraint graph starting at a variable, counting the number of times that a variable is reached during the traversal. This count is used to determine the order of constraint evaluation.

```
FormPlan(cs: ConstraintSeq; v: Variable)
(1)  for each c ∈ ConsumingConstraints(v), out ← c.output do
(2)     out.count ← out.count - 1
(3)     if out.count = 0 then
(4)        Append c to the end of cs
(5)        FormPlan(p, out)
```

`FormPlan` creates an ordered list of constraints to be executed through the use of the count value created by `SetCount`. The constraint graph is traversed topologically, appending each dependent constraint to the plan after its output variable is encountered for the last time.

```
ExecutePlan(cs: ConstraintSeq)
(1)  for each c ∈ cs in order do
(2)     Execute c.method
```

`ExecutePlan` executes the methods of each constraint in the plan according to the topological order determined in `CreatePlan`. This has the effect of computing new variable values for the downstream variables from the "inputs" parameter of `CreatePlan`. When a plan is executed, it is assumed that the structure of the constraint graph has not changed since the plan was formed. If constraints are added or deleted after a plan is created, then the plan should not be executed. Unpredictable behavior may occur if such a plan is executed.

The time complexity of `CreatePlan` and `ExecutePlan` is O(N), where N is the number of constraints. From `CreatePlan`, both `SetCount` and `FormPlan` traverse each constraint at most once since a "count" field of a variable is used to eliminate redundant traversals. `ExecutePlan` executes each constraint at most once since the created plan is the topological ordering of constraints from a given set of variables.

5.5 Optional extensions

This section describes some optional extensions to the UltraBlue algorithm for *limiting the computation direction* of constraints and *value consistency*, which allows inequalities and other kinds of invariants to be maintained.

5.5.1 Limiting computation direction

UltraBlue supports multi-way constraints, allowing a constraint to be dynamically redirected to compute any one of its associated variables. However, in many cases it is useful to limit a constraint so that only a subset of its associated variables may be an output of the constraint's computation. One-way constraints may be achieved by only allowing one variable of each constraint to be the output variable. For example, Section 4.1 uses an active value constraint to draw a graphics object when one of its attribute variables has changed. Drawing the object is a "side effect" of executing a constraint that is connected to its attribute variables. It does not make sense to redirect an active value constraint since this drawing side effect cannot be reversed.

This extension involves a few minor changes to the previous pseudocode. First, an additional field called "numOutVariables" should be added to the constraint structure. This represents the number of variables in the constraint's "variables" sequence that can be computed by the constraint. Second, function `AddConstraint` should be modified to take an additional parameter, `n`, which is used to set `numOutVariables`. The `variables` field of `Constraint` should be ordered with the computable variable at the beginning of the sequence. Third, functions `WalkStrength` and `SelectOutput` should be modified to only operate on the possible output variables according to `numOutVariables`. This extension does not change the time complexity of adding a constraint.

5.5.2 Value consistency

Typically, dataflow constraints are used to compute equation-based, equality relationships among a set of variables. In this case, the value of a variable is determined by possibly many upstream variable values in the constraint graph. The algorithm described thus far, as well as other dataflow algorithms, cannot manage inequalities and other assertions in a way that is consistent with the multi-way ability of constraints. While a constraint method is free to limit the range of its output variable values, this often results in upstream variable values which are inconsistent with downstream variable values.

This section describes *value consistency*, an extension to the UltraBlue algorithm that allows inequalities and general purpose verification methods on variable values. For example, Section 4.1 describes the constraint graph representation of a rectangular shape. This constraint graph includes a constraint on its width, left, and right variables. The width variable can be computed by subtracting the left value from the right value. Although users can freely manipulate the handles of a rectangular shape, the computed width should not ever be less than zero. This relationship can be maintained through the use of a value consistency assertion.

field	***type***	***initial value***	***description***
constraints	ConstraintSeq	∅	constraints to be computed
inputs	Set of Variables	∅	variables whose values will be changed externally

Table 3: ConsistencyPlan fields.

Table 3 describes the fields of the `ConsistencyPlan` structure that is used in creating and executing plans with value consistency assertions. Consistency is achieved by storing the plan's set of "input" variables whose values will be propagated by the plan (i.e., the `inputs` parameter supplied to

CreatePlan). When a plan is executed, the value consistency assertions of each downstream variable are executed to validate that the appropriate invariants are maintained. If there are invariants that are not satisfied, the plan's input variables are reverted sequentially in an attempt to satisfy all appropriate invariants; the constraint graph maintains existing valid value states. Note that the addition of constraints can cause existing variable values to become invalid, not satisfying all value consistency assertions.

field	*type*	*initial value*	*description*
oldValue	any type	any value	previous value of the variable
verifyMethods	Set of Assertions	user supplied	assertions on the value of the variable

Table 4: Additional fields for Variable.

Table 4 describes additional fields that need to be added to `Variable` in order to support value consistency. The `oldValue` field is used to store the previous value of a variable. When an "input" variable is assigned a new value, the current value should first be stored into the `oldValue` field. To revert a value, the `oldValue` field is copied to the `value` field.

```
CreateConsistencyPlan(inputs: Set of Variables) : VerifyPlan
(1)  p ← a new VerifyPlan
(2)  p.inputs ← {v | v ∈ inputs, v is editable}
(3)  p.constraints ← CreatePlan(inputs)
(4)  return p
```

`CreateConsistencyPlan` creates and returns an execution consistency plan.

```
ExecuteConsistencyPlan(p: ConsistencyPlan)
(1)  ExecutePlan(p.constraints)
(2)  count ← number of invalid variables in p
(3)  if count ≠ 0 then
(4)     VerifyValues(p, count)
```

`ExecuteConsistencyPlan` executes the constraint methods of a plan, minimizing the number of invalid variable values (if any) by calling `VerifyValues`. A variable value is invalid if one or more of its value consistency assertions are not satisfied.

```
VerifyValues(p: ConsistencyPlan; count: Integer)
(1)  mark ← a new unique mark
(2)  for each v ∈ p.inputs do
(3)     Revert v to its previous value
(4)     ExecutePlan(p.constraints)
(5)     newCount ← number of invalid variables associated with p
(6)     if newCount = 0 then
(7)        return
(8)     if newCount < count then
(9)        v.reverted ← mark
(10)       count ← newCount
(11)    else Revert v to its newer value
(12) for each v ∈ p.inputs such that v.reverted ≠ mark do
(13)    Revert v to its previous value
(14) ExecutePlan(p.constraints)
```

`VerifyValues` minimizes the number of invalid variable values computed by a plan. When called, there exists at least one such invalid variable value associated with the plan. Since the invalid variable value(s) may be caused by only one or a few of the plan input variables, each input variable is reverted separately. The loop in lines 2 - 11 individually reverts the value of a plan input variable, executes the

plan, and counts the number of invalid variable values. A variable value remains reverted if it reduces the total number of invalid variables. Note this is not an exhaustive approach of reverting all possible combinations of variable values; such an approach would result in a combinatorial time complexity. If invalid variables remain after the loop terminates, then all of the input variable values are reverted to their previous values.

Let M represent the cardinality of the "inputs" parameter supplied to `CreateConsistencyPlan`. Let N represent the total number of constraints in a constraint graph. The time complexity of `CreateConsistencyPlan` is O(max(M,N)) since it simply calls `CreatePlan` and filters a list of input variables. `ExecuteConsistencyPlan` takes O(MN) time, since it can potentially call `ExecutePlan`, taking O(N) time, for each of the M variables supplied in creating the plan.

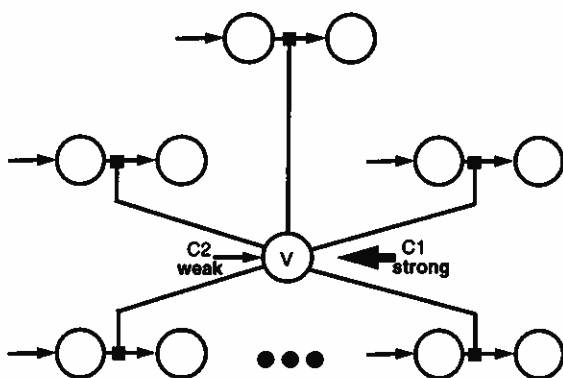
6 PERFORMANCE BENCHMARKS

This section compares the measured running time of the DeltaBlue algorithm to the UltraBlue algorithm using three constraint benchmarks. These benchmarks measure the time to add a new constraint to a constraint graph (the execution stage is not measured). DeltaBlue was chosen as the comparison algorithm since it is the most similar to UltraBlue. That is, it supports hierarchies of multi-way, single-output, dataflow constraints. Since DeltaBlue is unable to resolve cycles of constraints, cyclic benchmark comparisons were not possible.

Both the DeltaBlue and UltraBlue algorithms were implemented in C++ by the author using similar programming techniques. The benchmark programs were compiled using the GNU C++ compiler 2.6.3, optimization level 3, and were executed on a Sparc 20 workstation with the Solaris 2.4 operating system. The following timing measurements represent the total CPU running time of the DeltaBlue and UltraBlue algorithms with respect to *N*, the total number of constraints in each benchmark.

6.1 Star benchmark

The star benchmark adds a constraint to a central variable, V, that is referenced by every constraint in a star-shaped network (Figure 11). Each such constraint is connected to V and two other variables, one of which is constrained with a weak stay constraint. An application of this configuration is a constraint-based scaling factor, where the central variable represents a common scale and each constraint connected to V multiplies a value by the scale.



N	*DeltaBlue*	*UltraBlue*
10,000	113	52
20,000	225	101
30,000	335	153
40,000	448	199

Computation time in *milliseconds*.

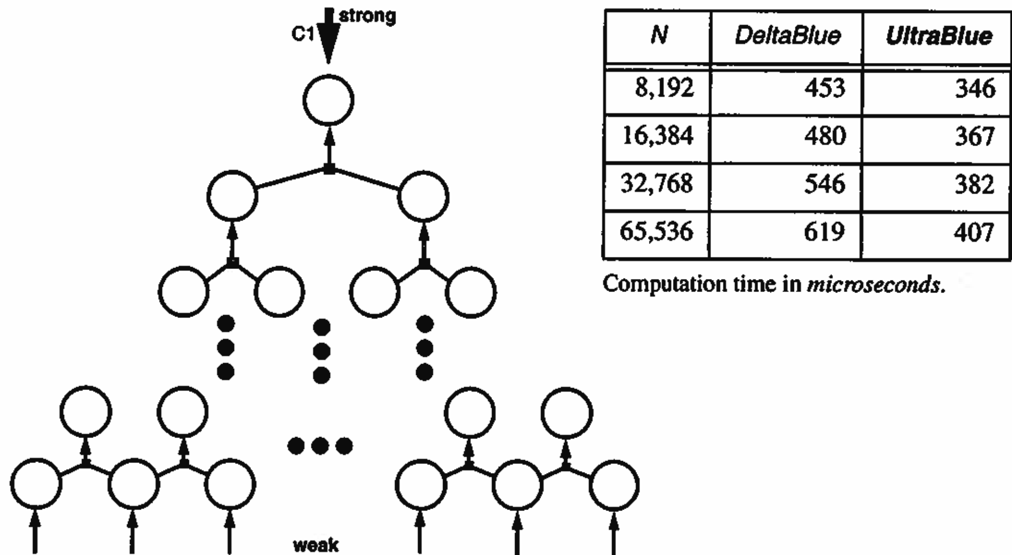
Figure 11: Star benchmarks and results.

The central variable of the star constraint graph has an associated weak stay constraint (C2). The star benchmark measures the time to add an additional strong constraint (C1) to the central variable. When constraint C1 is added, constraint C2 is overridden, and propagation of new strengths occurs to all downstream variables. As Figure 11's table shows, both DeltaBlue and UltraBlue execute this benchmark in

O(N) time, with UltraBlue running a factor of two faster than DeltaBlue. The performance difference is mainly due to the list-based strength propagation method of DeltaBlue.

6.2 Tree benchmark

The tree benchmark consists of a complete binary tree of required constraints (Figure 12). Each leaf constraint of the tree has an associated weak stay constraint.



N	*DeltaBlue*	***UltraBlue***
8,192	453	346
16,384	480	367
32,768	546	382
65,536	619	407

Computation time in *microseconds.*

Figure 12: Tree benchmark and results.

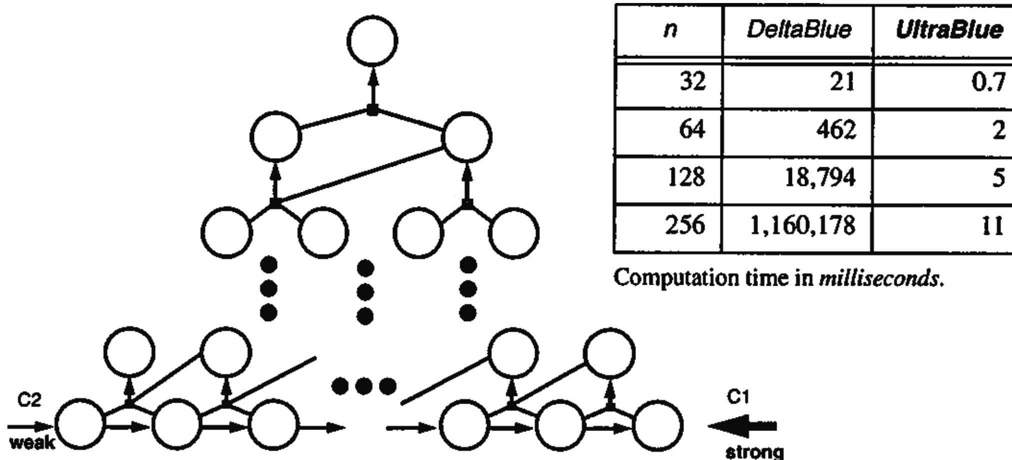
The tree benchmark measures the time to add a strong constraint (C1) to the root of the tree. This addition results in a redirection of constraints directly to a leaf of the tree, overriding the leaf's weak stay constraint (the choice of leaves is arbitrary). As Figure 12's table shows, both DeltaBlue and UltraBlue execute this benchmark in O(lg N), with UltraBlue running a constant factor faster than DeltaBlue.

6.3 Pyramid benchmark

A pyramid constraint graph is similar in appearance to a complete binary tree (Figure 13). In addition, for each inner constraint C such that there is an "uncle" variable U to the right, C is a four-way constraint among C's parent variable, C's children variables, and U. Also, each pair of consecutive "leaf" variables have a connecting equality constraint between them. The leftmost "leaf" has an associated weak stay constraint (C2), that has the effect of directing all equality constraints to the right. All constraints other than C1 and C2 are of required strength.

The pyramid benchmark measures the time to add a strong constraint (C1) to the rightmost "leaf" of the pyramid. This has the effect of redirecting each of the equality constraints among the "leaves," overriding C2, and propagating a new walkabout strength throughout the constraint graph. Figure 13's table shows the runtime performance of DeltaBlue and UltraBlue. UltraBlue executes this benchmark in O(N) time. DeltaBlue's running time is exponential due to its method of maintaining the variable walkabout strengths.

The time complexity of the DeltaBlue algorithm has been reported as O(N) [23]. However, that analysis made the assumption that every *underlying, undirected* constraint graph of is acyclic. The O(N) time complexity is not applicable to this benchmark since the underlying constraint graph of the pyramid is cyclic, even though the directed constraint graph is acyclic. The UltraBlue algorithm was created in order to support common user interface applications (Section 3) that inevitably involve constraints whose



n	*DeltaBlue*	***UltraBlue***
32	21	0.7
64	462	2
128	18,794	5
256	1,160,178	11

Computation time in *milliseconds.*

Figure 13: Pyramid benchmark and results.

underlying, undirected constraint graph is cyclic.

Since DeltaBlue is not able to resolve cycles of constraints, the pyramid's initial configuration for the DeltaBlue benchmarks had to be carefully constructed. That is, each constraint of the pyramid had to be added in a particular order so that directed cycles were not formed during pyramid construction.

SUMMARY

UltraBlue is an efficient incremental algorithm for satisfying hierarchies of multi-way, single-output, dataflow constraints through the use of local propagation. Contributions include a value consistency mechanism for maintaining arbitrary assertions (e.g., inequality relationships) and a cycle avoidance algorithm for resolving cyclic constraint relationships. Cycles of constraints are resolved with respect to each constraint's relative strength, making it possible to construct acyclic constraint graphs that can be efficiently solved, while preferring constraints of greater importance. While the general problem of cycle avoidance with this type of constraints is NP-complete, UltraBlue is a $O(DN^2)$ time heuristic algorithm (where D is the maximum constraint "fan-out" of a variable, and N is the number of constraints) that finds acyclic constraint graphs while preferring constraints with higher strength. The performance benchmarks show that UltraBlue out-performs the DeltaBlue algorithm on acyclic constraint problems.

UltraBlue is fast enough for user interface applications requiring real-time direct manipulation. It has been used as a basis of communication and interaction in the EUPHORIA user interface management system for more than a year. Several constraint applications in EUPHORIA were discussed, including end-user definable constraints, visualization of constraints, end-user specified preferential constraints, and integration of user interface components with interprocess communication. Cycle avoidance and the other features of the UltraBlue algorithm are needed to meet the diverse needs of this real world application. In practice, UltraBlue has proven itself to be extremely useful and versatile, greatly reducing the amount of programming involved in the creation of the EUPHORIA user interface management system. Future work includes enabling end-users to define arbitrary constraint relationships through the use of a calculator-like interface.

Information and on-line demonstrations of The Programmers' Playground and EUPHORIA are available on the World Wide Web [5].

ACKNOWLEDGMENTS

Ken Goldman was very helpful and encouraging, providing suggestions and listening to ideas and problems encountered during the development of the UltraBlue algorithm. David Saff implemented

constraint visualization and editing, giving EUPHORIA's users more understanding and control in working with consternated graphics objects. This research was supported in part by National Science Foundation grants CCR-91-10029 and CCR-94-12711, and ARPA contract DABT63-95-C-0083.

REFERENCES

[1] Alan Borning, and Bjorn Freeman-Benson, The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces. To appear in Proceedings of the First Incarnation Conference on Principles and Practice of Constraint Programming, September 1995.

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

[3] Bjorn Freeman-Benson, John Maloney, Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54-63, 1990.

[4] Michael Gleicher. A Graphics Toolkit Based on Differential Constraints. In *Proceedings of the 1993 ACM Symposium on User Interface Technology*, pages 109-120.

[5] Kenneth J. Goldman, et al. "Welcome to the Programmers' Playground!" http://www.cs.wustl.edu/cs/playground/

[6] Kenneth J. Goldman, T. Paul McCartney, Bala Swaminathan, and Ram Sethuraman. The Programmers' Playground: A Demonstration. In *Proceedings of the 1995 ACM International Conference on Multimedia*, November 1995. To appear.

[7] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications. *IEEE Transactions on Software Engineering*, 21(9):735-746, September 1995.

[8] J. Gosling, Algebraic Constraints. Ph.D. Dissertation, Carnegie-Mellon University School of Computer Science technical report CMU-CS-83-132, May 1983.

[9] Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. In *Proceedings of the ACM Symposium on User Interface Software*, pages 167-178, October 1988.

[10] Ralph D. Hill. Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *ACM Conference on Human Factors in Computing Systems*, pages 335-342, May 1992.

[11] Ralph D. Hill. The Rendezvous Constraint Maintenance System. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, pages 225-234, November 1993.

[12] Scott E. Hudson. A System for Efficient and Flexible One-Way Constraint Evaluation in C++. Georgia Institute of Technology College of Computing technical report 95-15, April 1993.

[13] Dan Ingalls, Scott Wallace, et al. Fabrik: A Visual Programming Environment. In *OOPSLA Conference Proceedings*, pages 176-190, September 1988.

[14] John H. Maloney. Using Constraints for User Interface Construction. Ph.D. Thesis, University of Washington Department of Computer Science and Engineering technical report 91-08-12, August 1991.

[15] T. Paul McCartney and Kenneth J. Goldman. EUPHORIA Reference Manual. Washington University Department of Computer Science technical report WUCS-95-19, July 1995.

[16] T. Paul McCartney and Kenneth J. Goldman. Visual Specification of Interprocess and Intraprocess Communication. In *Proceedings of the 10th International Symposium on Visual Languages*, pages 80-87, October 1994.

[17] T. Paul McCartney, Kenneth J. Goldman, and David E. Saff. EUPHORIA: End-User Construction of Direct Manipulation User Interfaces for Distributed Applications. To appear in *Software Concepts and Tools*.

[18] Olsen, D., Dempsey, E. and Rogge, R. Input/Output Linkage in a User Interface Management System. Com-

puter Graphics 19(3):191-197, July 1985.

[19] Mark W. Perlin. Reducing Computation by Unifying Inference with User Interface. Carnegie Mellon School of Computer Science technical report CMU-CS-88-150, June 1988.

[20] Michael Sannella. Constraint Satisfaction and Debugging for Interactive User Interfaces. Ph.D. Thesis, University of Washington Department of Computer Science and Engineering technical report 94-09-10.

[21] Michael Sannella. SkyBlue: A Multi-way Local Propagation Constraint Solver for User Interface Construction. In proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, pages 137-146, November 1994.

[22] Michael Sannella and Alan Borning. Multi-Garnet: Integrating Multi-way Constraints with Garnet. University of Washington Department of Computer Science and Engineering technical report 92-07-01, September 1992.

[23] 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, Vol. 32 No. 5, pages 529-566, May 1993.

[24] I, Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the Spring Joint Computer Conference*, pages 329-345, IFIPS 1963.

[25] Brad Vander Zanden. An Incremental Algorithm for Satisfying Hierarchies of Multi-way, Dataflow Constraints. University of Tennessee Computer Science Department technical report ut-cs-95-282, March 1995.

[26] Brad Vander Zanden. Incremental Constraint Satisfaction and its Application to Graphical Interfaces. Ph.D. Thesis, Cornell University, 1988.