Figure 3-1: Image morphing application in EUPHORIA and its output.*

image). The user specifies the correspondence between the two images by dragging the corresponding points to various locations on the images. For example, to establish a correspondence among the mouth positions, the widgets labelled "10," "11," "12," and "13" from the first aggregate mapping are positioned around Chewbacca's mouth and the widgets labelled "10," "11," "12," and "13" from the second aggregate mapping are positioned around Yoda's mouth.

A separately running "morphing module" performs the morphing operation. From the point information specified by the aggregate mappings, the morphing module constructs a collection of triangles that are used by the interpolation algorithm. The morphing module produces the series of intermediate metamorphosis images (i.e., a movie) as output. A separate EUPHORIA GUI (not shown) is used to specify the number of images and to display the resulting movie.
3.2.2 Distributed Minimum Spanning Tree

This application displays an animated visualization of the Gallager-Humblet-Spira distributed minimum spanning tree (MST) algorithm [16]. In this application, each vertex of the underlying graph is an independently running module (see Figure 3-5). The modules work together to find the minimum spanning tree of the complete graph according to the Euclidean distance among the vertices. Each vertex communicates with its adjacent vertices through message passing. Concurrently, tree fragments are formed by merging adjacent fragments, starting with single vertex fragments and ending with the minimum spanning tree. The user can arrange the vertices through direct manipulation to specify any arbitrary graph topology.

![Figure 3-2: Distributed minimum spanning tree in EUPHORIA.](image)

For example, Figure 3-2 shows a partially constructed minimum spanning tree consisting of 20 vertices forming three fragments. As described in Section 2.7, each vertex is a widget that displays its level number textually and displays its various states through the use of alternatives. Messages among the vertices are represented textually (e.g., “test(2, 129)”) and are animated.
between vertices over time. Both the vertices and messages are created through the use of aggregate mappings (Section 3.5). Since the edges are specified by the position of their vertices, the edges are created through the use of a joined aggregate mapping (Section 3.5.2) with the vertex aggregate.

3.3 Related Work

The purpose of a coordination language [17] is to separate communication from computation in order to offer programmers a uniform communication abstraction that is independent of a particular programming language or operating system. The separation of computation from communication permits local reasoning about functional components in terms of well-defined interfaces and allows systems to be designed by assembling collections of individually verified components. Coordination languages typically provide a structured configuration mechanism for specifying relationships among program modules. For example, Darwin [35] is a configuration language for managing message-passing connections between process ports in a dynamic system. Processes are expressed in a separate computation language that allows ports to be declared for interconnection within Darwin. Conic, the predecessor of Darwin, provides a graphical configuration mechanism for establishing bindings among the ports [34]. However, the modules of the system must still be concerned with when to send or receive messages on these ports. In Polylith [59], a configuration is expressed using "module interconnection constructs" that establish procedure call bindings among modules in a distributed system. CONCERT [83] provides a uniform communication abstraction by extending several procedural programming languages to support the Hermes [70] distributed process model. PROFIT [32] provides a mixture of data sharing and RPC communication through facets with data and procedure slots that are bound to slots in other facets during compilation. Extensions to PROFIT enable dynamic binding of slots in special cases [24]. The Weaves system [22] provides a configuration mechanism based on dataflow.

The ViewStation system [74] provides support for interactive media-based applications, where modules perform explicit communication using send and receive primitives. The VuSystem programming environment includes a set of programming conventions, media processing elements and a TCL-based [56] GUI for specifying both in-band media communication and out-of-band control
communication. User interfaces to these applications are typically constructed by writing TCL scripts. With the Visual Obliq system [3], dialog box style user interfaces are created using an interface builder. Distributed communication among the application and the user interface is accomplished by embedding callback code in an interpreted language.

The Rendezvous project [26], [28] concentrates on the separation of user interfaces from their applications through the use of interprocess communication. Rendezvous is a transition from purely user interface oriented systems to systems that decouple the construction of the graphical user interface from their applications. GUIs are constructed by creating programs using MEL, a language extension to Common Lisp providing support for graphics operations, object-oriented programming, and constraints. Constraints are used as Rendezvous' interprocess communication mechanism between an abstraction (controlling source code) and a view (its visualization). Playground does not utilize constraints for interprocess communication. Instead, constraints are used exclusively to define relationships between the attributes of graphics objects within a GUI. Playground's interprocess communication abstraction, logical connections, decentralizes the communication of modules in a distributed system; Playground modules communicate asynchronously without the need for a centralized constraint solver.

3.4 Data Boundary

A Playground module's data boundary consists of published variables that can be read and/or written from the external environment (see Section 1.5.1). When a published variable is changed in a module, Playground implicitly communicates the change out to all connected modules according to the module's associated logical connections. In this way, the computation of a module is decoupled from the application's communication structure as well as other modules.

EUPHORIA does not have separate modes for creating and running GUIs. Instead, GUIs can be created interactively at run-time and can be modified while they are being run. This gives the designer of a GUI a sense of instant gratification, since one can immediately see the results of changes during construction. This also gives end-users the ability to customize a GUI (possibly created by someone else) even while its application is running! The drawing command palette and
data boundary of the EUPHORIA window (see Figure 2-1) can be hidden, giving the end-user an unobstructed view of the GUI.

EUPHORIA was developed as a Playground module, enabling the end-user to expose certain GUI state components to EUPHORIA's data boundary as published variables. EUPHORIA supports all Playground base types (integer, real, character, boolean, string, memory block) as well as end-user defined tuples and aggregates. Published variables are graphically represented as color-coded rectangles (e.g., "TEMP" and "FIRE" in Figure 2-1), with the color representing the data type of each variable. Animated visualizations and interactive direct manipulation GUIs can be created by connecting appropriate attributes of a EUPHORIA drawing to external modules. End-users establish intraprocess communication between the data boundary and graphical components through the use of constraints (as in Figure 3-3). When the EUPHORIA module receives a change to one or more of its published variables, these changes are propagated simultaneously into the GUI's constraint graph (Section 3.6), resulting in changes to the GUI's appearance and/or animation. When a graphics object is manipulated by the user, manipulation changes are also propagated into the GUI's constraint graph, resulting in the modification of the published variables, and hence communication to external modules.

3.4.1 Tuples

Tuples provide a mechanism for treating a logically related collection of heterogeneous values as a single unit. In EUPHORIA, end-users can declare and instantiate tuple types dynamically at runtime. For more information, see Appendix B.

3

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3-3}
\caption{An end-user defined tuple.}
\end{figure}

For example, Figure 3-3 shows an end-user defined tuple representing coordinates of a rectangle. In Figure 3-3a, the tuple's fields are hidden. Figure 3-3b shows the tuple's fields hierarchically,
including a nested "point" tuple. Figure 3-3c shows the upper and lower fields connected to the handles of an image through the use of constraints.

### 3.4.2 Aggregates

An aggregate is a homogeneous collection of data values. Playground supports a number of aggregate data types including arrays and groupings. In EUPHORIA, end-users can declare and instantiate aggregates dynamically at run-time (for more information, see Appendix B). Aggregate types are used to construct aggregate mappings (Section 3.5).

![Diagram](image)

Figure 3-4: An end-user defined array.

For example, Figure 3-4 shows an end-user defined array of vertex tuples that was used in the distributed minimum spanning tree example (Section 3.2.2). Figure 3-4a-c shows the hidden and exposed graphical representations of this array and its element type.

### Element-to-Aggregate Connections

An element-to-aggregate connection results when a connection is formed between a data item of type T and an aggregate data item with element type T. For example, a client/server application could be constructed by having the server publish a data structure of type set(T) and having each client publish a data structure of type T. Playground supports element-to-aggregate connections, facilitating the construction of client/server applications and other types of applications. EUPHORIA allows aggregates to be involved in element-to-aggregate connections.

For example, the "nodes" aggregate from the distributed minimum spanning tree application (Section 3.2.2) gathers node information, published as individual tuples, from a number of
modules (Figure 3-5). This simplifies the construction of the application since the vertex modules can be connected directly to EUPHORIA without having a separate published variable for each vertex.

3.5 Aggregate Mappings

GUIs for large scale applications typically consist of a number of dynamically changing collections of graphical components. For example, the Image Morphing application in Section 3.2.1 has two such collections of graphical components: the 80 widgets associated with the corresponding points in the start and finish images. It would be very inconvenient to create these widgets and their corresponding published variables individually. Instead, it is desirable to treat these items as aggregates and automate their visualization.

An aggregate mapping is a mechanism for visualizing and manipulating the elements of an aggregate. End-users define an aggregate mapping by specifying constraint relationships among the aggregate's representative element fields (i.e., the element type fields) and a prototype instance of the visualization (e.g., a widget). For each element of the aggregate, the system creates a copy of the prototype instance and inserts it into the display. The copy is constrained to its corresponding element according to the specified relationships among the representative element and the prototype instance. The result is a collection of graphics instances whose attributes are associated with the aggregate's elements. This operation is functionally similar to the "project"
operation in relational databases since only a subset of the representative element's fields need to be used in the aggregate mapping.

![Diagram](image)

Figure 3-6: Aggregate mapping of an MST vertex.

For example, the vertices from the Distributed Minimum Spanning Tree example (Section 3.2.2) were created using an aggregate mapping. Figure 3-6a shows an array with a tuple representative element (i.e., Vertex) and a prototype instance (i.e., a vertex widget). In Figure 3-6b, the end-user defines constraints among the fields of the representative element and the prototype instance. The LN field is connected to the widget's text handle, the SN field is connected to the widget's alternative handle, and the pos field is connected to the widget's center handle. When the aggregate mapping is initiated, a copy of the vertex widget is created for each element of the array with the center, text, and alternative attributes constrained to the corresponding aggregate element fields.

### 3.5.1 Filtered Aggregate Mappings

It is often advantageous to view only a subset of an aggregate's elements. Filtering of extraneous elements results in a faster display that is easier to comprehend. For this reason, we have developed a mechanism that allows the end-user to filter the displayed elements based on a predicate. For example, one could specify the predicate of "FN < 100" on Figure 3-6's aggregate mapping, having the effect of only displaying vertices whose fragment number is less than 100. This filtering operation is functionally similar to the "select" operation in relational databases.
3.5.2 Joined Aggregate Mappings

Many times, single aggregates do not contain all of the relevant information needed to make a desired aggregate mapping visualization. Instead, information may be spread among multiple aggregates from distributed sources. This is especially true of an application that is constructed from off-the-shelf modules which were created by multiple programmers. With a joined aggregate mapping, the data of multiple aggregates is coordinated within an aggregate mapping based on a matching operation between key fields.

![Diagram showing joined Edges and Vertices aggregates.](image)

Figure 3-7: Joining Edges and Vertices aggregates.

For example, the Distributed Minimum Spanning Tree example (Section 3.2.2) uses an aggregate mapping to display the edges of the tree. To properly display an edge, it is necessary to know the positions of the edge’s two associated end-points. However, the Edges aggregate does not explicitly store this information. Instead, it stores a set of paired ID numbers v1 and v2 of the end-points; the vertex position information is stored separately in the Vertices aggregate. Joining the v1 and v2 key fields of Edges to the id key field of Vertices (Figure 3-7) creates virtual representations of the Vertex tuple within the Edges aggregate that can be used in establishing an aggregate mapping. For information on how to establish a joined aggregate, see Appendix B.
A joined aggregate is established among one or more supplier aggregates to a consumer aggregate. For example, Vertices is used twice as a supplier aggregate for the Edges consumer aggregate. For each value of the consumer aggregate, aggregate mapping instances are created based on matching key field values among the consumer and its suppliers. For example, if half of the vertices have id=0 and half have id=1, specifying an edge of (0, 1) would result in the visualization of a bipartite graph (i.e., an edge between each 0 vertex and each 1 vertex is created). In this way, joined aggregates are functionally similar to "join" and "cross product" operations of relational databases.

Matching Algorithm

The matches of a joined aggregate can change incrementally over time in several different ways. For example, key field values of one or more consumer and/or supplier aggregate elements can be changed at any time by the application. Also, values from consumer and/or supplier aggregates can be added or removed at any time by either the application or the user. For reasonable performance, the matching algorithm must be able to match efficiently, must support cross products efficiently, must keep track of existing matches to avoid redundancy, and must be able to quickly remove matches in a variety of ways.

\[
\{S_1, S_2\} \times \{S_3, S_4, S_5\} \times C_1 \\
\{S_1, S_3, C_1\} \\
\{S_1, S_4, C_1\} \\
\{S_1, S_5, C_1\} \\
\{S_2, S_3, C_1\} \\
\{S_2, S_4, C_1\} \\
\{S_2, S_5, C_1\}
\]

Figure 3-8: Cross product match of an aggregate with two joined fields.

Consider the process of enumerating a single element value's matches from the consumer aggregate. The matches can be viewed as an \(n+1\) term cross product, where the first \(n\) terms represent the supplier values matching each of the joined fields and the last term represents the consumer value. The result of this cross product is a series of tuples of size \(n+1\), each representing a match. For example, Figure 3-8 shows a cross product of a two-level join. The first join field
matches against values \( S_1 \) and \( S_2 \) of its supplier aggregate. The second join field matches against values \( S_3, S_4, \) and \( S_5 \) of its supplier aggregate. The result is a series of six match 3-tuples.

![Diagram](image)

Figure 3-9: Joined aggregate matching trie.

An \( n+2 \) level complete trie is used to maintain information about the current collection of match tuples. Each inner trie level represents the suppliers' matches to the joined fields. Each path from the root to a leaf represents a match \( n \)-tuple. For example, Figure 3-9 shows a trie for an aggregate with two joined fields. This example includes the first three match tuples of Figure 3-8's cross product as well as the match tuples of other cross products. The highlighted path represents the \( (S_1, S_4, C_1) \) match.

This method of storage has a number of advantages. First, adding a new match grouping is efficient. It is not necessary to check the entire collection of match groupings to see if a new match tuple already exists. Instead, a new match grouping is simply weaved into the trie. Second, for large cross product operations, this storage representation is space efficient as compared to storing each match grouping separately as a list. Third, removing a single match term is efficient. For example, suppose that \( S_1 \)'s key value is changed by the application. This means that it is necessary to remove all match groupings whose first term is \( S_1 \). This is achieved by simply deleting the sub-tree rooted at the \( S_1 \) node (and all associated graphical instance representations). Finally, deleting a graphic instance representation is efficient. This delete operation involves
deleting the nodes up the trie that are unique to the match tuple, starting at the appropriate leaf node.

3.6 Constraint Architecture

Figure 3-10 provides an overview of how both interprocess and internal communication occurs within EUPHORIA. A Playground application consists of a number of modules, including EUPHORIA, each of which have a set of published variables. Interprocess communication between modules is defined through logical connections among the published variables. Whenever a published variable is changed by its module, the change is communicated to the connected variables according to the logical connections.

![Diagram of EUPHORIA communication structure.]

When external values are communicated to EUPHORIA’s published variables, EUPHORIA reacts by copying the values into constraint variables associated with the data boundary. These constraint variables are connected to the constraint graphs of graphics objects within EUPHORIA (see Section 2.8.1). The values of these variables are propagated through the connected constraint graphs, having the effect of changing graphics object attributes according to the established
constraint relationships and redrawing graphics objects. In the same way, internal changes to graphics objects (e.g., direct manipulation) are propagated through the constraint graph and copied into EUPHORIA's published variables, sending the values out to external applications.

Graphics objects and EUPHORIA published variables also have associated ports, shown as tall rectangles in Figure 3-10, that are used to manage bundles of constraints. Ports and bundles are used for internal bookkeeping and type checking. For example, a rectangle has an upper left corner attribute that is of type "point." This attribute is actually represented as a pair of constraint variables for the x and y coordinates of the point and is managed by a port. This port cannot be connected to the port representing the width of a rectangle, since width is of type "real number." A bundle between "point" ports is actually a pair of constraints between the x and y values of the ports. Ports and bundles are also used to visualize constraints.

Bouncing Ball Example

An interactive bouncing ball could be created as follows. An oval representing the ball is drawn in EUPHORIA (Figure 3-11a). The oval is set to be a circle by constraining its width and height to be equal. 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 configured to a "Bouncer" module that simulates the effects of gravity over time as well as reacting to user interaction (Figure 3-11b). Additionally, a rectangle may be drawn (not shown) representing the boundary to which the ball can bounce. The top-left and bottom-right corners of this rectangle can be published and connected to the Bouncer module. The code for the Bouncer module is given in Appendix D.

Figure 3-12 shows the constraint graph for the bouncing ball 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 are 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.
The user may also drag and throw the ball through direct manipulation (Figure 3-13). 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.

3.7 Summary

In order to construct a complete distributed multimedia application, it is necessary to associate user interface components with their underlying application modules. This chapter describes end-user
mechanisms for exposing important state information from GUI to the external environment, allowing the modules of a distributed application to control and view the GUI’s state. Visualization and manipulation of dynamically changing collections of data is accomplished through the use of end-user defined aggregate mappings. Aggregate mappings include mechanisms similar to a relational database’s project, select, join, and cross product operations.
Chapter 4

UltraBlue Constraint Solver Algorithm

Multi-way constraint solvers have proven to be useful in the development of user interfaces and other applications [6], [15], [26], [40], [76]. However, previous such constraint solvers lacked the ability to express inequalities or to effectively handle cyclic constraint relationships [27], [65], [75]. 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 chapter describes UltraBlue, an efficient incremental algorithm for solving hierarchies of multi-way, single-output, dataflow constraints using local propagation. 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 constraint is NP-complete [39], UltraBlue is a O(\(D\times N\)) 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.
4.1 Contributions

1. An $O(DN^2)$ incremental, hierarchical, multi-way constraint solver algorithm.

2. A value consistency mechanism for maintaining arbitrary assertions (e.g., inequality relationships).

3. A cycle avoidance heuristic algorithm for eliminating cyclic constraint relationships.

4.2 Background

Much work has been done in the area of constraint maintenance. This section summaries basic terminology.

4.2.1 One-way and Multi-way Constraints

One-way dataflow constraint systems have been used extensively as a means of forming basic constraint relationships [27], [29], [54], [58]. 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 constraint always has the same input and output variables). 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 2.8)

4.2.2 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 [7], [15], [65] 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.

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 [15], 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 4-1: Adding a constraint to a hierarchical constraint graph.](image)

For example, Figure 4-1a 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 4-1b 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.

4.2.3 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 execution, 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.

![Diagram showing cycle solving versus cycle avoidance constraint systems.]

**Figure 4-2:** Example of cycle solving versus cycle avoidance constraint systems.

For example, Figure 4-2a 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 4-2b-d compares the effects of adding a strong constraint representing "B=C" between B and C using the DeltaBlue algorithm [65], 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 4-2b). 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 4-2c). 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 4-2d). 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 2.5). Unsolvable cycles
similar to Figure 4-2c 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.

4.3 Related Work

The SketchPad system [72] 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 [18] 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 Garnet [48], Fabrik [30], RENDEZVOUS [26], [27], and Hudson's incremental attribute evaluation algorithm [29]. However, this method of evaluation does not avoid unsolvable series of equations (see Section 4.2.3), and can leave constraint computations unsatisfied.

The DeltaBlue algorithm [15], [39], [65] 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 [15]. 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 4.2.3). 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 [39], 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 (Section 4.2.3). The Magritte system [23] 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 [63] provides support for solving hierarchies of multi-way, multi-output constraints, and has been applied in the Garnet toolkit [64]. 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 [75] 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.
4.4 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 4.4.1 - Section 4.4.3 describes the required portion of the UltraBlue algorithm for maintaining an acyclic graph of hierarchies of multi-way constraints. Section 4.4.4 describes an efficient algorithm for evaluating a series of constraints in topological order. Section 4.4.5 presents two optional extensions to UltraBlue for limiting the computation direction of constraints and value consistency enforcement.

4.4.1 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.

\[ \text{Weaker}(s1, s2 : \text{Strength}) : \text{Boolean} \]

(1) \textbf{return} true iff s1 is weaker than s2
4.4.2 Variable

Table 4-1 describes the variable structure, consisting of a number of fields for maintaining variable values, constraint graph topology, and other internal bookkeeping information.

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>initial value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>any type</td>
<td>user supplied</td>
<td>current value</td>
</tr>
<tr>
<td>constraints</td>
<td>Set of Constraints</td>
<td>Ø</td>
<td>constraints that reference this variable</td>
</tr>
<tr>
<td>determinedBy</td>
<td>Constraint</td>
<td>null</td>
<td>constraint that computes this variable, or null</td>
</tr>
<tr>
<td>walkStrength</td>
<td>Strength</td>
<td>weakest</td>
<td>walkabout strength</td>
</tr>
<tr>
<td>count</td>
<td>Integer</td>
<td>0</td>
<td>used for topological ordering</td>
</tr>
<tr>
<td>mark</td>
<td>Symbol</td>
<td>0</td>
<td>used for cycle detection</td>
</tr>
<tr>
<td>breakPoint</td>
<td>Constraint</td>
<td>n/a</td>
<td>used to find cycle breakpoints</td>
</tr>
</tbody>
</table>

PropagateStrength(v: Variable)
(1) for each $c \in$ 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.
4.4.3 Constraint

Table 4-2 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.

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>initial value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>VariableSeq</td>
<td>user supplied</td>
<td>variables referenced by this constraint</td>
</tr>
<tr>
<td>output</td>
<td>Variable</td>
<td>null</td>
<td>output variable of this constraint, or null</td>
</tr>
<tr>
<td>method</td>
<td>Method</td>
<td>user supplied</td>
<td>function to compute the value of an output variable</td>
</tr>
<tr>
<td>strength</td>
<td>Strength</td>
<td>user supplied</td>
<td>preference level in the constraint hierarchy</td>
</tr>
</tbody>
</table>

**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.

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.
\textbf{RemoveConstraint}(c: \text{Constraint})
\begin{enumerate}
\item \textbf{for each} \(v \in c.\text{variables} \) \textbf{do}
\item \( v.\text{constraints} \leftarrow v.\text{constraints} - \{c\} \)
\item \textbf{if} \(\text{Enforced}(c)\) \textbf{then}
\item \(\text{out} \leftarrow c.\text{output}\)
\item \(\text{Unenforce}(c)\)
\item \textbf{for each} \(c_1\) \textbf{such that } \(c_1\) \text{ is downstream from out}, \(\neg\text{Enforced}(c_1)\) \textbf{do}
\item \(\text{Enforce}(c_1)\)
\end{enumerate}

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.

\textbf{Unenforce}(c: \text{Constraint})
\begin{enumerate}
\item \(\text{out} \leftarrow c.\text{output}\)
\item \(c.\text{output} \leftarrow \text{null}\)
\item \(\text{out.\text{determinedBy}} \leftarrow \text{null}\)
\item \(\text{out.\text{walkStrength}} \leftarrow \text{weakest}\)
\item \(\text{PropagateStrength}(\text{out})\)
\end{enumerate}

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

\section*{Enforcing a constraint}

\textbf{Enforce}(c: \text{Constraint})
\begin{enumerate}
\item \textbf{if } \(\neg\text{Enforced}(c)\) \textbf{then}
\item \(\text{updateVars} \leftarrow \emptyset\)
\item \(\text{RedirectPath}(c, \text{null}, \text{updateVars})\)
\item \textbf{if} \(\text{Enforced}(c)\) \textbf{then}
\item \(\text{updateVars} \leftarrow \text{updateVars} \cup c.\text{variables}\)
\item \(\text{Execute the methods of updateVars' downstream constraints}\)
\end{enumerate}

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 ← ∅
(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 ← {v ∈ c.variables such that
(2) v ≠ prev, Weaker(v.walkStrength, c.strength})
(3) if candidates ≠ ∅ 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.
Breaking cycles

UltraBlue employs a heuristic algorithm for avoiding cyclic constraint relationships while attempting to satisfy the "locally-predicate-better" comparator (see Section 4.3). 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”
(7) mark c’s inputs with cmark
(8) b ← FindBreak(c, mark, cmark)
(9) if b ≠ null and b ≠ c then
(10) old ← b.output
(11) Unenforce(b)
(12) RedirectPath(b, old, updateVars)
(13) until b = null or b = c
(14) Reconnect c to its variables
(15) if b = c then
(16) old ← c.output
(17) c.output ← null
(18) old.determinedBy ← null
(19) RedirectPath(c, old, updateVars)
(20) updateVars ← updateVars ∪ {old}
(21) if v.determinedBy = null then
(22) v.walkStrength ← weakest
(23) else v.walkStrength ← WalkStrength(v.determinedBy)
(24) 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 - 20.

The loop in lines 5 - 13 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 \( \leftarrow \) null
(6) for each cl \( \in \) ConsumingConstraints(c.output) do
(7) back \( \leftarrow \) FindBreak(cl, mark, cycleMark)
(8) if back \( \neq \) null and (best \( = \) null or
(9) \hspace{1em} Weaker(WalkStrength(best), WalkStrength(back)) then
(10) \hspace{1em} best \( \leftarrow \) back
(11) c.output.breakPoint \( \leftarrow \) best
(12) c.output.mark \( \leftarrow \) 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.

**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 4.5.1).
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 $O(DN^3)$. 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 $O(DN^3)$ since the call to EliminateCycle in line 12 can be invoked recursively at most N times, executing in $O(DN)$ time.

In practice, the average 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 4.5.2).
4.4.4 Plan

Many times, the values in a constraint graph are updated repeatedly based on a fixed subset of changing variable values. For example, Section 2.8.3 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. 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. Also discussed is 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.

### 4.4.5 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. Both of these extensions are used by the EUPHORIA user interface management system.
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 2.8.1 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 and does not affect the time complexity of adding or removing a constraint. 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:

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

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:

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

SelectOutput(c: Constraint; prev: Variable) : Variable
(1) candidates ← {v ∈ c.variables such that
(2)   position(v) < numOutVariables,
(3)   v ≠ prev, Weaker(v.walkStrength, c.strength)}
(4) if candidates ≠ Ø then
(5)   return v ∈ candidates such that v.walkStrength is the weakest
(6) else return null
```
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. For example, if a constraint uses input variables A, with a value of 4, and B, with a value of 3, to compute an output variable C as their sum, then adding a simple restriction within the constraint to not let C’s value exceed 5 will lead to inconsistency. That is, A+B≠C since 3+4≠5.

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 2.8.1 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.

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>initial value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraints</td>
<td>ConstraintSeq</td>
<td>∅</td>
<td>constraints to be computed</td>
</tr>
<tr>
<td>inputs</td>
<td>Set of Variables</td>
<td>∅</td>
<td>variables whose values will be changed externally</td>
</tr>
</tbody>
</table>

Table 4-3: ConsistencyPlan fields.

Table 4-3 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.

Table 4-4: Additional fields for Variable.

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>initial value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldValue</td>
<td>any type</td>
<td>any value</td>
<td>previous value of the variable</td>
</tr>
<tr>
<td>verifyMethods</td>
<td>Set of Assertions</td>
<td>user supplied</td>
<td>assertions on the value of the variable</td>
</tr>
</tbody>
</table>

Table 4-4 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.

### 4.5 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, not including the execution stage time. The execution stage time is not measured since it varies based on the nature of the application’s computation, independent
of the constraint solver. However, if it were included, there would be even more of a difference in the benchmark times (i.e., in UltraBlue's favor) since DeltaBlue also executes its constraints while finding a conflict-free constraint graph. 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.

### 4.5.1 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 4-3). 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.

![Figure 4-3: Star benchmarks and results.](image)

<table>
<thead>
<tr>
<th>N</th>
<th>DeltaBlue</th>
<th>UltraBlue</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>113</td>
<td>52</td>
</tr>
<tr>
<td>20,000</td>
<td>225</td>
<td>101</td>
</tr>
<tr>
<td>30,000</td>
<td>335</td>
<td>153</td>
</tr>
<tr>
<td>40,000</td>
<td>448</td>
<td>199</td>
</tr>
</tbody>
</table>
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 4-3'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.

4.5.2 Tree Benchmark

The tree benchmark consists of a complete binary tree of required constraints (Figure 4-4). Each leaf constraint of the tree has an associated weak stay constraint.

![Tree benchmark and results](image)

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 4-4's table shows, both DeltaBlue and UltraBlue execute this benchmark in O(lg N), with UltraBlue running a constant factor faster than DeltaBlue.
4.5.3 Pyramid Benchmark

A pyramid constraint graph is similar in appearance to a complete binary tree (Figure 4-5). 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 left-most "leaf" has an associated weak stay constraint ($C_2$), that has the effect of directing all equality constraints to the right. All constraints other than $C_1$ and $C_2$ are of required strength.

<table>
<thead>
<tr>
<th>$n$</th>
<th>DeltaBlue</th>
<th>UltraBlue</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>21</td>
<td>0.7</td>
</tr>
<tr>
<td>64</td>
<td>462</td>
<td>2</td>
</tr>
<tr>
<td>128</td>
<td>18,794</td>
<td>5</td>
</tr>
<tr>
<td>256</td>
<td>1,160,178</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 4-5: Pyramid benchmark and results.

The pyramid benchmark measures the time to add a strong constraint ($C_1$) to the right-most "leaf" of the pyramid. This has the effect of redirecting each of the equality constraints among the "leaves," overriding $C_2$, and propagating a new walkabout strength throughout the constraint graph. Figure 4-5'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 $O(N)$ [65]. However, that analysis made the assumption that every underlying, undirected constraint graph 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 that inevitably involve constraints whose 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.

4.6 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 two years. 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 EUPHORIA in comparison to hard-coding graphical relationships within the source code.
Chapter 5

Application Management

Distributed multimedia applications consist of multiple components (e.g., application portion, GUIs) with varying configurations and performance requirements. To use a completed application, this information must be used to ensure correct and optimal performance. Application management refers to a software approach for automating the process of launching and configuring complete distributed multimedia applications. With such a software system, end-users can use complex distributed multimedia applications without knowing the internal details of the application. For example, an application may have the requirement that a certain module must operate on a particular file system in order to access a locally stored database. Exposing this information directly to the end-user is undesirable since it forces the end-user to be knowledgable about the application’s internal details and the current performance characteristics of the resources that may be used. Instead, it is better to leave the launching details to an automated software system.

Using the techniques described in Chapter 1 - Chapter 3, end-users can configure modules of a distributed application and can create GUIs for the application. This process involves:

1. launching the Playground run-time system,
2. launching each application module using the UNIX shell,
3. creating appropriate logical connections among the modules,
4. drawing one or more GUIs, and
5. connecting state components of the GUIs to the appropriate application modules.
Although this is achievable by end-users, it is undesirable to build an application from scratch each time that it is used. Applications consist of modules running on different workstations over the Internet, making the launching operation rather time consuming. End-users will want to use completed applications that were constructed by someone else without having to separately launch each of the application’s modules or bother with forming logical connections among the modules. This chapter describes an application management system for The Programmers’ Playground that automates the process of launching and configuration. This system serves to make completed distributed multimedia applications available to a broad class of users and facilitate the development and use of multi-user applications.

5.1 Contributions

1. An application-independent architecture enabling automated processor allocation of long-lived distributed application modules, exploiting the separation of the communication structure from computational components inherent in the I/O Abstraction programming model.

2. Parameterized configurations enabling the automated configuration of complete distributed applications and client/server applications.

3. A World Wide Web service to maintain application information (parameterized configuration, performance requirements, etc.), enabling end-users to launch, configure, and join distributed multimedia applications on the Internet.

5.2 Motivating Examples

This section describes two example distributed applications that benefit from the use of application management.

5.2.1 Medical Image Processing Pipeline

A nuclear medicine radioactive blood pool study is used to create movies of the human heart for diagnostic purposes. Each movie consists of a series of images. However, the images suffer from noise caused by ambient radiation, making them difficult to read. This noise can be reduced by digitally filtering the images.