Constraints: An International Journal, 3, 1–26 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics

ALAN BORNING borning@cs.washington.edu
Department of Computer Science & Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350

BJORN FREEMAN-BENSON bnfb@oti.com
Object Technology International Inc., R. Buckminster Fuller Laboratory, 201 - 506 Fort St., Victoria, BC,CANADA V8W 1E6

Abstract. Ultraviolet is a constraint satisfaction algorithm intended for use in interactive graphical applications. It is capable of solving constraints over arbitrary domains using local propagation, and inequality constraints and simultaneous linear equations over the reals. To support this, Ultraviolet is a hybrid algorithm that allows different subsolvers to be used for different parts of the constraint graph, depending on graph topology and kind of constraints. In addition, Ultraviolet and its subsolvers support plan compilation, producing efficient compiled code that can be evaluated repeatedly to resatisfy a given collection of constraints for different input values.

Keywords: constraints, user interfaces, hybrid constraint satisfaction algorithms

1. Introduction

Many key aspects of interactive graphical systems can be conveniently described using constraints, including layout and other kinds of geometric relations, consistency between application data and views, consistency of multiple views, and animation. In selecting the kinds of constraints to be supported in a user interface toolkit or other interactive graphical system there are various tradeoffs between simplicity and power. Some of these tradeoffs are:

- numeric constraints only, or other domains allowed as well
- one-way or multi-way constraints
- functional constraints only, or more general kinds of relations
- required constraints only, or constraint hierarchies
- acyclic constraint graphs only, or cycles allowed

Numeric constraints are the mainstay of interactive graphical applications. However, other constraints are useful as well, such as constraints on colors, strings, or fonts. A constraint is *one-way* if each constraint has a distinguished output variable, and if the solver is only allowed to change that variable to satisfy the constraint, while a constraint is *multi-way* if in general the solver is free to change any of the constraint's variables to satisfy that constraint. A constraint is *functional* if, for each of its constrained variables v not annotated as read-only, there is a unique value for v that will satisfy the constraint, given values for the other variables. A *constraint hierarchy* is a set of constraints labelled by strengths. The constraints labelled as *required* must be satisfied, while those labelled with weaker strengths are merely preferences [4]. One important application of constraint hierarchies is in representing our desire that parts of a graphical object don't move unnecessarily, by placing weak stay constraints on them. This allows us to give a simple declarative semantics for constraint satisfaction in the presence of state and change over time. We can also use constraint hierarchies for expressing other kinds of preferences, for example, that a figure track the mouse if possible (but not if it bumps up against an immoveable object). The final tradeoff concerns the constraint graph rather than individual constraints. We can view the constraints and constrained variables as forming a bipartite graph. Each variable and each constraint is represented by a node, with an edge from a constraint node to a variable node if the variable is constrained by the constraint [11]. The constraint graph is acyclic iff this bipartite graph is acyclic.

All of these properties are declarative attributes of the constraints or the constraint graph, rather than of the constraint satisfaction algorithm. Naturally, however, the different properties place different requirements on the algorithm. One class of algorithm that has been explored by a number of researchers, and used in a variety of systems, is that of local propagation algorithms for multi-way constraints, constraint hierarchies, functional constraints only, and no cycles. These algorithms provide a good balance between expressiveness and efficiency.

In local propagation algorithms, each constraint has a set of methods that can be used to satisfy the constraint. When a method is executed, it sets one of the constrained variables to a value such that the constraint is satisfied. The algorithm itself imposes no restriction on the domain of the variables — constraints over domains such as strings, colors, and so forth can be accommodated in a straightforward fashion as long as the local propagation methods are provided. However, all of the constraints in a traditional local propagation algorithm must be functional, since otherwise one couldn't provide the methods for that constraint.

There are two major limitations of local propagation algorithms. First, they don't support cycles (for example, constraints representing simultaneous linear equations). Our experience has been that most UI problems can be represented using acyclic constraint networks — problems that intrinsically involve simultaneous equations are not so common. However, cycles do often arise when the programmer adds redundant constraints, and it is an added burden for the programmer to avoid this. Further, it is clearly contrary to the spirit of the whole enterprise to require programmers to be constantly on guard to avoid cycles and redundant constraints; after all, one of the goals in providing constraints is to allow programmers to state what relations they want to hold in a declarative fashion, leaving it to the underlying system to enforce these relations. The second major limitation is that standard local propagation algorithms don't support inequality constraints. Inequality constraints are useful in a variety of user interface applications, particularly in layout. For example, we might want a constraint that one window be to the left of another, or that a figure be contained within a rectangle. However, since inequality constraints aren't functional, they aren't supported by traditional local propagation algorithms. Consider the constraint $a \leq b$: given b we can't uniquely determine a.

The Ultraviolet algorithm is intended to address these two limitations. It can handle simultaneous equations, acyclic collections of inequalities, and some collections of simultaneous linear inequality constraints, while still supporting constraints over arbitrary domains. To accomplish this, Ultraviolet is actually a hybrid algorithm, which supports a number of subsolvers, including ones for traditional local propagation, for interval-based local propagation, for simultaneous linear equations, and an incomplete solver for simultaneous linear inequalities. Ultraviolet has been implemented in OTI Smalltalk.

This paper represents a snapshot of an ongoing research project on using constraints in interactive graphical applications, and there have been a number of significant recent developments, which as of this writing have not been reflected in the Ultraviolet implementation. The conclusion (Section 7) includes some discussion of the strengths and weaknesses of the algorithm, based on experience with the algorithm and in light of these new developments, and suggests avenues for future work.

2. Related Work

There is a long history of using constraints for interactive graphics. The first constraint-based system, as well as the first interactive computer graphics system, was Ivan Sutherland's Sketchpad [31] from 1963, a constraint-based drawing system. Sketchpad provided multi-way constraints and interactive response, and paved the way for many subsequent systems.

Many constraint-based systems have employed one-way constraints; recent examples of toolkits and algorithms that use one-way constraints are Amulet [27], subArctic [21], and Hudson's ultra-lightweight constraint system [22]. The other most common sort of constraint satisfaction algorithm for interactive graphics is a local propagation algorithm for multi-way constraints (thus restricting the constraints to functional constraints only and acyclic constraint graphs). Recent examples include DeltaBlue [30], SkyBlue [29], and QuickPlan [32]; these three algorithms also support constraint hierarchies.

The DETAIL system [20] is quite closely related to Ultraviolet. Like Ultraviolet, it is designed for user interface applications, and is a kind of meta-solver that supports different subsolvers. It supports multi-way constraints and constraint hierarchies, using local propagation when possible, and grouping constraint cycles into cells, which are then solved by an appropriate subsolver. The most recent version of DETAIL [19] also includes an experimental solver for inequality constraints as well as functional constraints.

Ultraviolet uses the Indigo algorithm [1] for acyclic collections of inequality constraints. Indigo is an interval propagation algorithm; there has been considerable work on interval constraints in other areas of computer science, particularly artificial intelligence and constraint logic programming. Davis [9] discusses the completeness and running time for interval propagation algorithms, as well as for other kinds of labels. Hyvönen [23, 25, 24] presents a number of interval constraint satisfaction algorithms, and also describes the generalization of interval propagation to division propagation. (A division is a union of ordered, non-overlapping intervals.)

Two algorithms that provide efficient support for simultaneous linear equality and inequality constraints are QOCA and a more recent variation named Cassowary [17, 6]. Both algorithms are derived from the simplex algorithm. They maintain the constraints in a solved form that allows new solutions to be computed rapidly as input variables are changed, and also allow efficient incremental addition and deletion of constraints. The major difference between the two algorithms is that QOCA finds least-squares solutions (i.e. solutions that minimize the squares of the errors of conflicting constraints), while Cassowary finds locally-error-better solutions (see Section 3 for a definition of locally-error-better). A toolkit that uses the QOCA algorithm has been used in a diagram parser and editor [8] and for layout of trees and graphs [16], while Cassowary has been used in a web authoring tool [5].

Other systems that allow for inequality constraints and cycles often use numeric algorithms. Notable examples are Juno [28], Juno-2 [18], and Briar and Bramble, which use a differential solver [12, 13, 14]. An earlier version of Ultraviolet — which doesn't include support for inequality constraints — is described in [3].

3. Constraint Hierarchy Solutions

As described in the introduction, constraint hierarchies allow both required and preferential constraints. A *solution* to a constraint hierarchy is a mapping from variables to domain elements. Given a constraint hierarchy, if not all of the preferential constraints can be satisfied, we need a way to select which solutions are desired. In our previous work on DeltaBlue and SkyBlue we've used the *locally-predicate-better* comparator, in which we are concerned only whether or not a constraint is satisfied in a given solution. This comparator has proven quite satisfactory for functional constraints. However, for inequality constraints, an alternative comparator, *locally-error-better*, is superior. (As an aside, while the specification for DeltaBlue and SkyBlue is that the algorithms produce locally-predicate-better solutions, in fact for these particular algorithms we can prove that the solutions produced are locally-error-better as well.)

We give a brief, informal description of these comparators here; for a formal definition see reference [4]. We will need to consider the error in satisfying a constraint. This error is 0 if and only if the constraint is satisfied, and becomes larger the further away the solution is from a satisfying one. For example, the error in satisfying the constraint $a + b = c$ is $|a + b - c|$. For constraints over domains that aren't metric spaces, we typically use a simple 0/1 error function that returns 0 if the constraint is satisfied and 1 if it is not.

A solution S is locally-error-better if there is no other solution T that is better than S. Informally, T is better than S if there is some level k in the hierarchy such that the errors for all the constraints in levels 0 to $k-1$ are exactly the same for T and S, and at level k the errors in satisfying each constraint using T are less than or equal to the errors using S, and strictly less for at least one constraint. There may be more than one locally-error-better solution to a given hierarchy. Ultraviolet's task is to find one of these solutions (not all).

Locally-predicate-better is the same as locally-error-better, except that we use the simple 0/1 error function for all constraints, including numeric ones.

To illustrate why locally-error-better gives more satisfactory results for inequality constraints in UI applications, consider an object constrained to lie within a fixed rectangle. Suppose the user is moving the object with the mouse, and tries to move it outside the rectangle. We'll make the mouse movement constraint strong but not required, so that the object will stop moving if it bumps up against an immovable obstacle, rather than giving an error. Using locally-predicate-better, if the user moves the object slowly, it will move as far as the side of the rectangle and then stop. However, if the user moves the mouse quickly (so that at one time the cursor is well within the rectangle and the next time outside), the figure will remain at the old location and not bump up against the side at all. (Since we can't satisfy the mouse constraint exactly, using locally-predicate-better we don't try to satisfy it at all.) Further, if the user tries to move the figure along the side of the rectangle it won't move unless the user gets the cursor positioned just on the boundary (but not outside it).

In contrast, with locally-error-better, the figure will follow the cursor until the cursor moves outside the rectangle. After that, the figure will move along the wall of the rectangle so that it is as close to the cursor as possible, as if the object were magnetically attracted to the cursor.

4. Architecture of the Ultraviolet Algorithm

As described in Section 1, one of our goals for the solver is to support local propagation constraints over arbitrary domains, as well as inequality constraints and simultaneous linear equality and inequality constraints over the reals. Devising an efficient, single solver for this range of constraints and constraint network types is difficult if not impossible. Instead, Ultraviolet supports multiple cooperating subsolvers, with a flexible architecture that allows new subsolvers to be introduced. The subsolvers currently available are:

- a local propagation solver for functional constraints (Blue)
- a local propagation solver for constraints on the reals, including inequalities (Indigo)
- a solver for simultaneous linear equations (Purple)
- a solver for simultaneous linear equations and inequalities (Deep Purple)

These subsolvers are discussed in Section 5. In the remainder of this section we describe the overall architecture of the Ultraviolet algorithm, including how constraints are sent to different subsolvers and how the subsolvers communicate.

As described in the introduction, we can view the constraints and constrained variables as forming a bipartite graph. The first step is to partition this graph into connected subgraphs. Each connected subgraph is solved independently.

Each connected subgraph is partitioned into cyclic and acyclic regions. The acyclic regions are further partitioned into regions containing numeric constraints and regions containing non-numeric constraints. We can view each region as itself a node in a larger graph; the overall graph will be acyclic. Each region has a marker indicating whether it is cyclic or acyclic, and a second marker indicating whether it is numeric (with inequality constraints and perhaps other numeric constraints as well), numeric (with functional constraints only), or non-numeric. Each constraint belongs to exactly one region; however, a variable can be shared by two or more regions. The subsolvers communicate via these shared variables.

To partition the connected subgraphs, regions containing similar constraint types are sorted by read-only annotations. The constraint types could be determined by asking each class of subsolver whether it can handle each constraint, but for efficiency we encode the standard types into the constraints. The types we encode are: numeric/non-numeric; linear/non-linear; equality/inequality. We also add the meta-type “cyclic” to cyclic sub-graphs. Read-only annotations are considered to break sub-graphs, so that constraints on each side of the read-only variable will be placed in separate regions. After this gathering phase, each region can be accurately labeled with the set of subsolver classes that could be used to solve the region.

This choice of regions and markers is motivated by the capabilities and efficiencies of the different subsolvers. We prefer to use a local propagation solver (Blue) when possible, since it is simple, efficient, and applies to constraints over arbitrary domains, not just numeric ones. However, this solver only works on acyclic networks of functional constraints. For acyclic regions that include inequality constraints, we need to propagate intervals rather than just values, but we can only propagate intervals through numeric constraints. Indigo does the job here. For cycles, if all the constraints are linear equalities, we can use Purple; if the cycle includes inequalities, we must use the less efficient Deep Purple subsolver. The remaining possibility is a cycle containing nonlinear or non-numeric constraints — if there is such a cycle the constraint graph is too difficult for our current collection of subsolvers, and in this case an exception is raised.

As noted above, when solving the connected subgraph the subsolvers communicate via the variables that are shared by two or more regions. These variables can either be unbound, or can hold a specific value (which will of course be the value for that variable in the eventual solution). In addition, for real-valued variables, when using inequalities the variable can also have bounds on it, which may be repeatedly tightened during constraint solving. Constrainable variables are represented as instances of a separate ConstrainableVariable class, providing a place to store this additional information as well as a specific value for the variable when known.

The obvious way to invoke the subsolvers would be to invoke them one at a time, letting each subsolver run to completion on its constraints. However, this would interact poorly with constraint hierarchies, since a weak constraint in region A might attempt to set a shared variable before a strong constraint in region B on the same variable had been considered. A key insight — due to Richard Anderson — is that instead we should invoke each solver in turn to process its required constraints, then each solver to process the strongest non-required constraints, and so on through all levels in the hierarchy. After each solver is invoked for a given strength, we must also propagate any changes to the variables to other solvers. This technique is well-suited for finding locally-error-better solutions — but it would not work for finding e.g. least-squares solutions, since it inherently considers some constraints of a given strength before others with the same strength, rather than determining some aggregate measure.

Here is the interface that must be implemented by each subsolver.

begin_solving(region)
: Initialize this subsolver with the constraints and variables in region.

solve_level(strength)
: Initialize the set of changed variables. Process the constraints at strength strength. Put any variables that have been set while processing these constraints, or whose bounds have been tightened, into the set of changed variables.

changed_variables
: Return the set of variables changed while processing the current strength, either by setting a value or tightening a bound. This set is initialized to the empty set by the solve_level method when it begins execution.

process_changed_variable(var)
: Variable var has been changed by another subsolver — propagate any changes that result within this region.

done_solving
: All levels in the hierarchy have been processed — do any necessary finalization.

Table 1 gives a pseudocode description of the main procedure solve. It uses an auxiliary procedure propagate_changes, shown in Table 2. The propagate_changes procedure takes two arguments: a particular solver that forms the starting point for the propagation, and a list of all solvers in the region. It considers the set of variables for the starting solver that have been changed while processing the current strength. This set is returned by the changed_variables message. The procedure then activates in turn each other solver that shares one of these variables (using the process_changed_variable message), asking this other solver to do any other processing that might be enabled by the new information. This may cause the new solver to change some of its variables, which may cause additional solvers to be activated, and so forth.

4.1. An Example

Here is a simple example. Consider the constraints shown in Table 3. We first partition the constraints and variables into regions, as shown in Table 4.

```
procedure solve(all_constraints,all_variables);
    /* partition the constraints and variables into connected subgraphs */
    connected_subgraphs := find_subgraphs(all_constraints,all_variables);
    for subgrph in connected_subgraphs do
        /* partition subgraph into regions */
        regions := find_regions(subgrph);
        /* select and initialize a subsolver for each region */
        solvers := emptyset;
        for region in regions do
            solver := select_solver(region);
            solver.begin_solving(region);
            solvers := solvers $\cup$ {solver};
        end for;
        strengths := list of all constraint strengths in subgrph, strongest first;
        for strength in strengths do
            for solver in solvers do
                solver.solve_level(strength);
                propagate_changes(solver,solvers);
            end for;
        end for;
    end for;
end procedure solve;
```

Table 1. Ultraviolet pseudocode.

```
procedure propagate_changes(root_solver,all_solvers);
    solver_queue := queue.new;
    variable_queue := queue.new;
    activate_solvers(root_solver,all_solvers,solver_queue,variable_queue);
    while solver_queue is not empty do
        s := solver_queue.first;
        v := variable_queue.first;
        s.process_changed_variable(v);
        activate_solvers(s,all_solvers,solver_queue,variable_queue);
        solver_queue.dequeue;
        variable_queue.dequeue;
    end while;
end procedure propagate_changes;

procedure activate_solvers(s1,all_solvers,solver_queue,variable_queue);
    for v in s1.changed_variables do
        for s2 in all_solvers do
            if s1 $\neq$ s2 and v $\in$ s2.variables then
                solver_queue.add(s2);
                variable_queue.add(v);
            end if;
        end for;
    end for;
end procedure activate_solvers;
```

Table 2. Ultraviolet pseudocode — auxiliary procedures.

Table 3. Example constraints.

required	$a = 10 * x$
required	$b = 5$
required	$c = a * b$
required	$x + y = 7$
required	$f = y + 12$
required	$h = 2 * g$
required	$s = printstring(h)$
strong	$3 * x + 5 * y = 21$
medium	$5 * x + 8 * y = 10$
medium	$g = 10 * f$
medium	$a = 0$

Table 4. Partitioning into regions.

Region	Variables	Constraints	
1	a, b, c, x	*required*	$a = 10 * x$
		required	$b = 5$
		required	$c = a * b$
		medium	$a = 0$
2	x, y	*required*	$x + y = 7$
		strong	$3 * x + 5 * y = 21$
		medium	$5 * x + 8 * y = 10$
3	y, f, g, h	*required*	$f = y + 12$
		required	$h = 2 * g$
		required	$s = \mathit{printstring}(h)$
		medium	$g = 10 * f$

Note that x is shared by Regions 1 and 2, and y is shared by Regions 2 and 3. Region 1 and 3 are both acyclic functional, and will be handled by instances of Blue, while Region 2 is a cyclic region containing linear equality constraints to be handled by a Purple solver.

We then invoke each solver in turn to process just its required constraints. The Blue solver for Region 1 processes its required constraints, using the $b = 5$ constraint to determine that $b = 5$. The other two constraints, $a = 10*x$ and $b = 5*a$, remain on its active list, since they cannot yet be enforced given the currently available information. The Purple solver for Region 2 processes the *required* $x + y = 7$ constraint, making x a nonparametric variable defined by the expression $-y+7$ (see Section 5.3). However, no variables are given values, so again the propagate_changes procedure doesn't do anything. Finally the Blue solver for Region 3 processes its required constraints; again no values can be determined, and all of its constraints are put on the active list.

Next we process the *strong* constraints for each region. The only region with constraints at this strength is Region 2. The $3*x+5*y = 21$ constraint allows the solver to find values for x and y, namely $x = 7$ and $y = 0$. The propagate_changes procedure is executed, and the newly determined value for x is sent to Region 1's solver, which is able to use it to determine that $a = 70$ and $c = 350$. The value $y = 0$ is then sent to Region 3's solver, allowing it to determine that $f = 12$.

Finally we process the *medium* constraints. The Blue solver for Region 1 processes the $a = 0$ constraint. Since a already has a value and this constraint isn't required, it is discarded. The Purple solver for Region 2 processes the $5 * x + 8 * y = 10$ constraint, replacing x and y by their values. This yields $35 = 10$, which is a contradiction, but again since this constraint isn't required it is discarded. No changes are propagated to other regions. The Blue solver for Region 3 processes the $g = 10 * f$ constraint. This allows the solver to determine that $g = 120$, $h = 240$, and $s =$ '240.0' using the active constraints. Since none of these variables are shared with other regions, no changes are propagated to other regions. (In a typical interactive application, all of the variables would have very weak stay constraints specifying that they should keep their old values. In this example, the weak stays would have no effect because all variables were given values by stronger constraints.)

4.2. Remarks

In the above example it was essential to have different solvers for some of the constraints: a simultaneous linear equation solver would not be able to handle the string constraint on h and s, while a local propagation solver would not be able to handle the simultaneous equations involving x and y. However, Regions 1 and 2 could have been collapsed into a single region and just handled by Purple. In this simple case, in fact that would probably be more efficient rather than having the overhead of two communicating solvers. However, for large local propagation networks keeping them separate may be more efficient. This is a question that should be investigated empirically. Note that in any case which choice is made doesn't affect the correctness or capabilities of the solver; it is purely an efficiency issue.

This example illustrates why all the required constraints (in all subsolvers) are processed first, then all the strong constraints, and so forth, rather than processing all of the constraints for one subsolver first, then all of the constraints for another. If we had processed all of the constraints in Region 1 before considering Region 2, the *medium* $a = 0$ constraint would have been used to deduce that $b = 0$ and $x = 0$ — but both these valuations would have been incorrect, since stronger constraints in Region 2 dictate otherwise. (There is in fact always *some* order in which the subsolvers could be invoked to allow each of them to run to completion before invoking any other subsolver — but finding this ordering may not be trivial. The current scheme is much more straightforward.)

When inequalities are involved, the subsolvers cooperate in much the same manner; the only difference is that we may communicate tightened bounds through the shared variables in addition to specific values.

5. Subsolvers

In this section we provide an overview of each of the current subsolvers. Each subsolver is a subclass of an abstract superclass Solver, which defines the message protocol for all subsolvers that was described in Section 4. The primary focus of this paper is the overall architecture of Ultraviolet and how the subsolvers communicate, rather than the details of each subsolver, so these are primarily outlines.

5.1. Blue

Blue is the simplest of the current subsolvers; it is a batch version of our incremental DeltaBlue algorithm [10, 30]. It is a traditional local propagation solver, which handles acyclic constraint graphs containing functional constraints only. However, it can process constraints over arbitrary domains, rather than just numeric constraints as with the other subsolvers. Each constraint that may be processed by Blue must have a collection of local propagation methods. Given a constraint *cn*, each of *cn*'s variables that is not annotated as read-only must have an associated method that can calculate a value for that variable that will satisfy the constraint, given values for the other variables. (Constraint *cn* must be functional, i.e. there must be only one such value.) For example, the constraint $a + b = c$ has three methods: `c := a+b`, `a := c-b`, and `b := c-a`. Executing any of these methods will result in the $a + b = c$ constraint being satisfied.

Blue's task is to determine which methods should be used, and in what order, to find values for the variables in its region. An instance of Blue keeps a queue of constraints to be processed, and also a dictionary of active constraints, each indexed by the variables to which the constraint applies. (Active constraints are constraints for which we are not yet able to find values for all of the constrained variables; these constraints must be considered again if one of their constrained variables is set by some other constraint, in case a propagation is possible.)

As with the other subsolvers, Blue is invoked to process the constraints at a particular strength using the `solve_level` method. Each constraint with the given strength is processed in turn. If enough information is available to deduce the value of one of the constraint's variables, this is done. We also check the dictionary of active constraints to see if any further propagations are possible. Otherwise, the constraint is indexed in the dictionary of active constraints. If a constraint sets the value of a variable shared by other regions, this variable is put into the `changed_variables` set, so that other subsolvers can be alerted.

When a constraint *cn* is being processed, if all of its variables already have values then *cn* cannot be enforced. If *cn*'s strength is *required*, Blue checks to make sure that *cn* is in fact satisfied and raises an exception if it is not; otherwise *cn* is simply discarded.

5.2. Indigo

Indigo solves acyclic collections of numeric constraints, which may include inequality constraints as well as functional constraints. As with Blue, Indigo is invoked to process the constraints at a particular strength. As each constraint with the given strength is processed, the permissible intervals for variables are tightened. If a constraint sets the value of a variable shared by other regions, or tightens its bounds, the variable is put into the `changed_variables` set.

Indigo is similar in many respects to Blue; the primary difference is that Indigo propagates intervals rather than values. Each constraint that can be processed by Indigo has a collection of bounds propagation methods. For example, the $a+b=c$ constraint has three bounds propagation methods, which tighten the bounds on a, b, and c respectively:

```
a.tighten(c.bounds - b.bounds)
b.tighten(c.bounds - a.bounds)
c.tighten(a.bounds + b.bounds)
```

If we have previously tightened the bounds on say c, when we process the constraint $a+b=c$ we may then need to tighten the bounds on both a and b. This is in sharp contrast to the behavior of standard local propagation algorithms, in which to satisfy a constraint a single method is executed (and hence a single variable changed).

Like Blue, Indigo maintains a queue of constraints to be processed and a dictionary of active constraints, each indexed by the variables to which the constraint applies. When Indigo is invoked to process the constraints at a given strength using the solve_level method, each constraint is considered in turn. The queue is initialized with the current constraint. We then process each constraint in the queue, tightening the bounds in its variables, and adding other constraints to the queue on these same variables (thus causing tightenings to ripple out through the constraint graph). This continues until the queue is empty. We keep track of any variables whose bounds have been tightened in the changed_variables set. A complete description of the algorithm is given in reference [1]; proofs of correctness theorems are given in [2].

5.3. Purple

Purple solves collections of constraints that can be represented as linear equations. Unlike Blue and Indigo, Purple is not troubled by cycles (i.e. simultaneous equations). The algorithm is adapted from that used in CLP($\mathcal{R}$) [26] and other Constraint Logic Programming language implementations [7], the only difference being provision for both required and preferential constraints. Variables are in one of three sets: *parametric*, *nonparametric*, and *known*. Parametric variables may take on any value. Nonparametric variables are defined by linear expressions involving parametric variables plus a constant. Known variables are ones known to be equal to a constant. (Known variables are in fact just a special case of nonparametric variables; for convenience we keep track of them separately.) Initially all variables are parametric.

Purple is invoked to process the constraints at a given strength using solve_level. Each constraint is converted to a linear normal form (or delayed if this is not possible). Nonparametric and known variables are replaced by their defining expressions, and the result is simplified and put back into normal form. If the result is $0=0$, the new constraint is implied by the existing constraints that have already been satisfied, and can be discarded. If the result is $0=c$ for some nonzero c, the constraint is incompatible with the existing constraints. If its strength is required, a required_constraint_not_satisfied exception is raised; if not the constraint is simply discarded. Otherwise, the constraint adds some new information (and can be satisfied). One of the equation's variables is selected as a new nonparametric variable and moved from parametric to nonparametric variables (or to known variables if the equation has only one variable). In addition, the new nonparametric variable is replaced by its defining expression in all of the expressions for other nonparametric variables. This may cause other variables to be moved from nonparametric to known variables. As usual, any variables whose values are set, i.e. variables that have been moved to known variables, are put into the set changed_variables. (However, variables that have been moved to nonparametric variables are *not* put into changed_variables, since a nonparametric variable can still take on any value, given suitable choices of values for the parametric variables that define it.)

For a multiplication constraint $a * b = c$ to be converted to linear normal form, at least one of a or b must be known at the time the constraint is processed. If neither a nor b is known, the constraint is instead added to the delayed_constraints set. Each delayed constraint is then re-checked after processing each weaker level, to see if it has become linear; if it has, it can then be processed. We check that each such delayed constraint can be satisfied when it is processed (even if it isn't at the required strength) — if it can't be satisfied, we raise a constraints_too_difficult exception, indicating that the constraints were too difficult for the solver to handle. The idea is that we *might* have been able to satisfy it, with a different choice of values for its constrained variables, but we weren't able to determine what those values should have been. We also raise a constraints_too_difficult exception if it never becomes linear. (This case will not normally arise in our interactive graphical applications, since usually we put very weak stay constraints on every variable to try and maintain it at its previous value; these weak stay constraints would give specific values for each variable if a value had not been determined earlier.)

5.4. Deep Purple

Deep Purple solves collections of linear equality and inequality constraints, including simultaneous equations and inequalities. We want to satisfy non-required constraints as well as possible to find locally-error-better solutions. In its full generality, Deep Purple's task is equivalent to a multi-objective linear programming problem — each required constraint maps to a constraint in the linear program, and each non-required constraint maps to an objective function that the error expression for the constraint have value 0. Stronger constraints map to objective functions with higher priority; ties among constraints with the same strength are broken arbitrarily. (Different ways of breaking these ties give rise to different but still correct locally-error-better solutions.) A naive implementation based on this observation would not have acceptable performance for interactive graphical use. Instead, the current algorithm is a fast and sound, but incomplete, solver, which solves a useful subset of the set of problems of this class.

We actually use two different techniques, first trying one, and if it fails, then the other. The first technique is to rewrite the collection of constraints into a collection of arbitrary required n-ary equational constraints and unary required and preferential constraints. It is straightforward to do this: for any non-required n-ary constraint at some strength s, or required inequality, we rewrite it into an equivalent pair consisting of an n-ary required equality constraint and a unary constraint at strength s. For example, *strong* $a \geq b$ is rewritten as *required* $a - b = u$ and *strong* $u \geq 0$, where u is a new variable. We then process just the required equality constraints using Purple. If the implicit constraint graph relating the parametric and nonparametric variables is acyclic, we produce required constraints corresponding to these relations. We add in any required unary inequalities and preferential unary constraints. (Note that the constraint graph remains acyclic.) This collection of constraints is then solved by Indigo.

If this technique fails because the implicit constraint graph is cyclic, we fall back on a second technique. Here we use Purple to solve the constraints, augmenting it by maintaining bounds on each variable. Unary inequality constraints are processed by tightening the bounds on the constrained variable appropriately; if the constraint can't be satisfied exactly given the previous bounds, it is satisfied as well as possible. (As usual, if a required constraint can't be satisfied we raise an exception.) Equality constraints are processed as with Purple, except that when a variable is moved to known_variables, we check that the newly found value is compatible with the bounds; if it is not, a constraints_too_difficult exception is raised. To process a binary inequality constraint (e.g. $x \leq y$), we tighten the bounds on its constrained variables. If after tightening bounds one or both variables still don't have specific values, we place the constraint on the active constraints list, and recheck it when the variables do take on specific values. (If it isn't satisfied at that point we raise a constraints_too_difficult exception.)

5.5. Incompleteness

Two desirable properties for a constraint solver are soundness and completeness. Ultraviolet and its subsolvers are all sound — that is, if they produce an answer, that answer will be a locally-error-better solution to the constraints. However, each of the subsolvers is incomplete in one or more ways, that is, there are collections of constraints that it cannot solve. (If this happens, we raise a constraints_too_difficult exception.)

Blue is complete if each local propagation method can always calculate a value for its variable that will satisfy the constraint, given values for the other variables. This is the case for most constraints — an exception being multiplication and division, caused by attempting to divide by 0 in a local propagation method. If this occurs, a runtime exception is raised, which is re-raised as a constraints_too_difficult exception.

Similarly, Indigo is complete if each propagation method calculates exact bounds for its variable given exact bounds for the others. It is easy to calculate exact bounds for most constraints, except for multiplication and division if the intervals include 0 — in this case, the calculated bounds are too large. If this occurs, as the computation continues, other intervals may be too large as well, and if eventually the algorithm attempts to narrow the interval for a variable to a specific value, the error will be detected when an active constraint is determined to be unsatisfiable, and a constraints_too_difficult exception will be raised. (This exception would also be raised if some variables did not have specific values after all the constraints have been processed. As noted in Section 5.3, normally this situation won't arise given very weak stays on each variable.) For a more detailed discussion of this issue see [2].

Incompleteness manifests in Purple when a constraint *cn* must be delayed, and after subsequent assignments of values to its variables by weaker constraints, we find that *cn* is unsatisfied.

For Deep Purple, incompleteness is manifested when the constraint graph cannot be transformed to an acyclic one. In this case, binary inequality constraints are simply checked but not enforced, so that if they happen to not be satisfied, incompleteness results. Another case is with unary inequality constraints for which the bounds should be propagated through equality constraints.

In practice, the incompleteness of Blue, Indigo, and Purple has never been observed as a problem in interactive graphical applications. Deep Purple works reasonably well when the constraints can be transformed into an acyclic constraint graph and be processed by Indigo. However, if the user includes redundant binary inequality constraints this transformation is not possible, and the resulting constraints are often not solvable. When we tested Ultraviolet on practical industrial applications, we found that in fact users often did include such redundant constraints. Thus to make Ultraviolet practical for a wide range of realistic applications, a complete solver for simultaneous inequalities is needed that replaces Deep Purple. Ways of addressing this need are discussed in Section 7.

6. Compilation

A common action in interactive graphical applications is to move a part of a complex object with the mouse. Each time the screen is refreshed for a new mouse position, the constraints must be re-satisfied; however, typically only the input values are changing, not the constraints themselves or their topology. To support good interactive performance in this situation, constraint systems for user interface applications have often supported some form of planning or compilation.

To support compilation in Ultraviolet we introduce a new kind of constraint, an *edit constraint*. Semantically an edit constraint acts just like a constant constraint $x = c$, i.e. some constrained variable x is constrained to be equal to a constant c at some strength. However, the value of c isn't known at compilation time. Rather, the compiler's task is to produce a plan containing a Smalltalk block (function), along with other information, so that the plan can be executed repeatedly to resatisfy the constraints with different values of c. An arbitrary number of edit constraints can be used; typically the plan includes two edit constraints attached to a point being moved, one for the x coordinate and one for the y coordinate.

A series of different plans, derived from different collections of constraints, can be precompiled and saved. This allows us to achieve the effect of rapidly switching between different collections of constraints. Often the difference between the collections will be just a different pair of edit constraints (corresponding to a move action for a different part of the figure), but more drastic changes in the collection of constraints are equally possible.

Constrainable variables maintain additional state to describe their status during compilation. A constrainable variable can be in one of the following states:

unknown The value of this variable is completely unconstrained, given the constraints processed so far.

compile_time_known The value of this variable is known at compile time.

run_time_known The value of this variable will be known at run time, but isn't known at compile time.

compile_time_bounded There are bounds on this variable (which are known at compile time), but it does not yet have a specific value.

run_time_bounded There will be bounds on this variable at run time, given the constraints processed so far, but not necessarily a specific value. These bounds aren't known at compile time.

This information is used by the subsolvers so that they can generate more efficient code. For example, suppose that variables x and y are marked as unknown, and we have processed a constraint $2*x=y$ (without being able to infer any more specific information about x or y). If we next process the constraint $x=5$, we know that x can safely be set to 5 and y to 10 — and furthermore, we know this information at compile time, since it is independent of information coming in from an edit constraint at run time. The compiled code can therefore simply include assignment statements to set x to 5 and y to 10. On the other hand, if we have a constraint *c edit*, and c is marked as unknown, then after processing the edit constraint c is marked as run_time_known — we don't know what c's value is at compile time, but we know it will have a specific value at runtime after satisfying the edit constraint.

We do similar reasoning for inequality constraints. If x is marked as unknown and y is marked as run_time_known, then after we process the constraint $x \geq y$ we mark x as run_time_bounded — we won't know a specific value for x at this point at runtime, but we will have some bounds information (and thus won't be free to set x to any arbitrary value, in contrast to variables marked as unknown).

Using this status information, we may be able to determine some values at compile time rather than at run time. In other cases, we may be able to avoid compiling code since we can determine that executing that code would have no effect.

The overall operation of the Ultraviolet compiler is similar to that of the solver described in Section 4. The interface implemented by each subsolver to support compilation is also similar to the interface for interpreted constraint satisfaction. Each subsolver must support the following invocations:

begin_compiling(region)
Initialize compilation for this subsolver.

compile_level(strength)
Initialize the set of changed variables, and compile code to satisfy the constraints at strength strength if possible. Put any variables that have been set while processing these constraints, or whose bounds have been tightened, into the set of changed variables.

changed_variables

Return the set of variables changed while processing the current strength, either by setting a value or tightening a bound. This set is initialized to the empty set by the compile_level method when it begins execution.

compile_process_changed_variable(var)

Variable var has been changed by another subsolver — compile code propagate to any changes that result within this region.

done_compiling

All levels in the hierarchy have been processed — do any necessary finalization.

6.1. Blue

The operation of the Blue compiler is similar to that for the solver. When a constraint is processed, if its variable v is marked as unknown and all its other variables are marked as compile_time_known, the local propagation method for v is used to set v to its appropriate value, and v is marked as compile_time_known. Otherwise, if v is marked as unknown and all its other variables are marked either as compile_time_known or run_time_known, then the constraint adds code to the method being constructed to set v (at run time) to its value, and v is marked as run_time_known. If an edit constraint can be enforced (because its variable is unknown), Blue generates code to set the variable at runtime to the value, which will be supplied to the plan each time it is run, and the variable is marked as run_time_known.

6.2. Indigo

The operation of the Indigo compiler is similar to that for the solver, and bounds propagation method compilation is handled in a manner analogous to that described for Blue. The principal novel feature of the Indigo compiler is the use of degrees of freedom information for variables to allow code for some of the bounds propagation methods to be omitted or more efficient code to be compiled. The analysis makes use of the status flags unknown, compile_time_known, run_time_known, compile_time_bounded, and run_time_bounded listed above. It is done on a constraint-by-constraint basis, and is handled by appropriate methods invoked for each constraint.

Consider the constraint $a + b = c$. Suppose that the bounds on c have just been tightened. Normally we would need to compile code to tighten the bounds on both a and b, which in turn would result in code to tighten bounds on variables connected to a or b by other constraints, and so forth, rippling out through the constraint graph. However, if both a and b are marked as unknown, so that the bounds for each are $(-\infty, \infty)$, then we know at compile time that tightening the bounds would have no effect, and so no code is compiled. Now suppose that a is marked as unknown but b as run_time_bounded. In this case we only compile code to tighten the bounds on a, since we know at compile time that no tightening for

b's bounds will occur. If a variable is marked as having a run_time_known value, we know that tightening its bounds would have no effect, and so no code is compiled in this case. Finally, suppose instead that c has a known value (i.e. its bounds have just been tightened to a single value), and b already has a known value. In this case we can compile code for the more efficient local propagation method as used by Blue to set the value of a.

To illustrate, consider the following constraints (all in an Indigo region).

required	$a + b = c$
required	$c \leq 10$
strong	*a edit*
weak	*a stay*
weak	*b stay*
weak	*c stay*

The variables a, b, and c are initially all marked as unknown. Processing the $a + b = c$ constraint leaves all of them still unknown — we could set any single one of them to an arbitrary value and still satisfy the $a + b = c$ constraint. We next process $c \leq 10$. This compiles code to tighten c's bounds to $(-\infty, 10]$. In general Indigo would then tighten the bounds on both a and b. However, since both are marked as unknown, we know either one can take on any value at this point, and so trying to tighten the bounds on a and b would have no effect — so no code is compiled for this.

We next process *strong a edit*. Since a is unknown, we know it can be set to the input value from the edit, whatever it may be, and so we simply compile an assignment statement to set a rather than the more general and expensive operation of tightening the bounds on a. We can now mark a as run_time_known. As before, in general Indigo would then tighten the bounds on both b and c. However, since b is still unknown, we know that tightening the bounds on c would have no effect — so we only compile code to tighten the bounds on b, and mark b as run_time_bounded.

We then process the weak stay constraints. We first process the stay on a — but since a is already marked as run_time_known it already has a specific value, and so this stay has no effect. Processing the stay on b, we do have to compile code, since b is run_time_bounded. We can now mark b as run_time_known — after processing the stay, b will be set to the single value within its current bounds that is closest to its old value. As usual, in general we would then tighten the bounds on both a and c — we do need to compile code to tighten the bounds on c, but since we know a already has a specific value, we don't need to compile code to tighten a's bounds. Since a and b had specific values, we know that tightening the bounds on c will result in a specific value for c, and so we can also now mark c as run_time_known. Finally, we process the stay on c, and since c is run_time_known, no code need be compiled.

The complete code for this example is as follows:

```
1.  c.bounds := (−∞, 10];
2.  a := input value;
3.  b.tighten(c.bounds-a.bounds);
```

```
4. b.tighten(b.old_value,b.old_value);
5. c.tighten(a.bounds+b.bounds);
```

Suppose that we execute this code with an input value of 5 for a, and that previously the variables had the values $a = 2$, $b = 8$, and $c = 10$. After executing line *1*, c's bounds will be $(-\infty, 10]$. After executing *2*, a will have value 5, i.e. its bounds are $[5, 5]$. After executing *3*, we will tighten b's bounds to $(-\infty, 5]$. Executing *4* will set b to 5, that being the closest value within $(-\infty, 5]$ to the desired value of 8. Finally, executing *5* sets c to 10. The result is thus $a = 5$, $b = 5$, $c = 10$, which is a locally-error-better solution to the constraints.

6.3. Purple and Deep Purple

In Purple, variables are partitioned into one of three sets: parametric, nonparametric, and known, as described in Section 5.3. The primary difference between the operation of the Purple solver and the Purple compiler lies in the treatment of edit constraints and variables marked as run_time_known. A variable may be marked as run_time_known when Purple processes an edit constraint. Also, a variable may be marked as run_time_known by another subsolver — in this case the Purple compiler is notified of this action by the compile_process_changed_variable invocation. When a variable is marked as run_time_known, it is locked in the parametric variable set by the compiler — it cannot be moved to nonparametric variables, or of course to known variables (which contains variables whose values are known at compile time).

When the Purple solver finishes processing all the constraint strengths, normally all variables will be in the known variable set. In contrast, for the Purple compiler, variables will be in the known variable set if their values are known at compile time; but some variables may remain in the sets of parametric and nonparametric variables. However, each remaining parametric variable should be run_time_known, and each remaining nonparametric variable should be a linear combination of parametric variables, each of which is run_time_known. The code generated by Purple simply sets each nonparametric variable to the appropriate value, given the values for the parametric variables that define it.

In more detail, suppose the Purple compiler processes an edit constraint on a variable v. If v is marked as run_time_known, it has already been given a value and so the edit constraint can't be enforced. If the edit constraint is required, we compile a runtime check to make sure that v's value is the same as that supplied by the edit constraint; otherwise we simply discard the edit constraint. Suppose instead that v is marked as unknown. If v is parametric, it can take on any value (and in particular it can take on the value that will be supplied at runtime for the edit constraint). We mark v as run_time_known. If v is nonparametric, we swap it with a parametric variable w in v's defining expression, making v parametric and w nonparametric, and mark v as run_time_known. Next we check if any of the nonparametric variables whose defining expressions involve v are now run_time_known — this is the case if all of the parametric variables in its defining expression is marked as run_time_known.

Table 5. Compilation example.

required	k *edit*	/* new constraint */
required	$a = 10 * x$	
required	$b = 5$	
required	$c = a * b$	
required	$x + y = k$	/* changed from the example in Section 3 */
required	$f = y + 12$	
required	$h = 2 * g$	
required	$s = printstring(h)$	
strong	$3 * x + 5 * y = 21$	
medium	$5 * x + 8 * y = 10$	
medium	$g = 10 * f$	
medium	$a = 0$	

If this is the case, we mark these nonparametric variables as run_time_known as well, and generate code to set them to their values.

The other case is when a variable v is marked as run_time_known by another subsolver. This case is handled similarly. If v is parametric, we leave it in the set of parametric variables, and check whether any nonparametric variables are linear combinations of run_time_known variables. If v is nonparametric, we swap it with a parametric variable, as described above.

The compiler for Deep Purple is similar, except that bounds are maintained for variables, and runtime checks are compiled to make sure that any attempt to set a variable to a value is within its current bounds.

6.4. An Example

Consider the example shown in Table 5. (This is similar to the example shown in Table 3 — the changes are commented in the table.) We partition the constraints and variables into regions as showin in Table 6.

The Blue compiler for Region 1 processes its required constraints, and determines that $b = 5$. (Note that b is thus compile_time_known.) However, it is unable to determine the values for a, c, or x, or to determine that any of them will be known at runtime. The Purple compiler for Region 2 first processes the *required k edit* constraint. All the variables in this region, including k, are initially in parametric variables, so k is marked as run_time_known and code is compiled to set k to the supplied input value at runtime. Purple then processes the *required* $x + y = k$ constraint, making x a nonparametric variable defined by the expression $k - y$. (y could have been selected as the new nonparametric variable instead, but not k.) Finally, the Blue compiler for Region 3 processes its required constraints, but no values can be determined.

Next we process the *strong* constraints for each region. The only region with constraints at this strength is Region 2. We process the $3*x+5*y = 21$ constraint, first substituting the expression $k-y$ for x in the constraint and then simplifying to yield $3 * k + 2 * y = 21$. Purple chooses y as a new nonparametric variable (it can't

Table 6. Partitioning the constraints for compilation.

Region	Variables	Constraints	
1	a, b, c, x	*required*	$a = 10 * x$
		required	$b = 5$
		required	$c = a * b$
		medium	$a = 0$
2	x, y, k	*required*	k *edit*
		required	$x + y = k$
		strong	$3 * x + 5 * y = 21$
		medium	$5 * x + 8 * y = 10$
3	y, f, g, h	*required*	$f = y + 12$
		required	$h = 2 * g$
		required	$s = printstring(h)$
		medium	$g = 10 * f$

choose k, since k has been marked as run_time_known). It solves for y, yielding the equation $y = -1.5*k+10.5$, and adds y to nonparametric variables. The expression $-1.5 * k + 10.5$ is also substituted for the occurrence of y in the expression for x, so that x is now defined by the expression $2.5 * k - 10.5$. Both of the defining expressions for the nonparametric variables x and y now involve only variables marked as run_time_known, so the compiler marks x and y as run_time_known as well, and adds the following assignment statements to the compiled code:

```
x := 2.5*k - 10.5;
y := -1.5*k + 10.5;
```

The propagate_changes procedure is executed. Blue for Region 1 is notified that x is now run_time_known using the compile_process_changed_variable message. Blue compiles the following assignments based on the local propagation methods for its constraints, and marks a and c as run_time_known:

```
a := 10*x;
c := a*b;
```

Also, Blue for Region 3 is notified that y is run_time_known, so it compiles the following assignment and marks f as run_time_known:

```
f := y+12;
```

Finally we process the *medium* constraints. The Blue compiler for Region 1 processes the $a = 0$ constraint. Since a is already marked as run_time_known, this constraint can't be enforced, but since it isn't required it is discarded. The Purple compiler for Region 2 processes the $5*x+8*y = 10$ constraint, replacing x and y by their defining expressions. This yields $k = -43$. Since k is already run_time_known, we can't assign another value to k, but since the constraint isn't required it is discarded. No changes are propagated to other regions. The Blue compiler for

Region 3 processes the $g = 10 * f$ constraint. This allows Blue to add the following assignments to the generated code and to mark g, h, and s as run_time_known:

```
g := 10*f;
h := 2*g;
s := printstring(h);
```

Since none of these variables are shared with other regions, no changes are propagated to other regions.

The complete code is:

```
initial values: b=5;

k := input value;
x := 2.5*k - 10.5;
y := -1.5*k + 10.5;
a := 10*x;
c := a*b;
f := y+12;
g := 10*f;
h := 2*g;
s := printstring(h);
```

To use this plan for several different input values for k, we set $b = 5$ (just once), then run the code for each input value. For example, for $k = 7$, we get $x = 7$, $y = 0$, $a = 70$, $c = 350$, $f = 12$, $g = 120$, $h = 240$, and $s =$ '240.0'.

7. Conclusion

This paper represents a snapshot of an ongoing research project on using constraints in interactive graphical applications. In the time since this paper was written, there has been both experience in using Ultraviolet as well as a number of significant new developments. In this section we discuss the strengths and weaknesses of the algorithm, and suggest directions for future work.

We experimented with using Ultraviolet in a number of industrial applications, primarily for diagram layout and manipulation in systems for computer-aided software engineering. Regarding the power of the solver, we found that users often added redundant constraints, since this arose naturally when expressing the desired layout. For linear equality constraints this is no problem, but for inequality constraints it often resulted in cycles that Deep Purple was unable to solve. Based on this experience we concluded it is essential to provide a complete solver for simultaneous linear inequalities. Regarding performance, the speed of the compiled code was excellent. However, the time to run the compiler was a problem if the collection of constraints changed repeatedly as a result of adding and deleting parts of the diagram.

There are now several candidates for complete solvers for simultaneous inequalities intended for user interface applications. Two of these, QOCA and Cassowary

(Section 2), are adaptations of the simplex algorithm. They are both incremental and have very good performance. Their disadvantages are that they only handle linear equalities and inequalities, and unlike the Ultraviolet compiler, a runtime solver is needed. Another disadvantage is that the update time is variable: while the average time is very fast, if there are many non-required constraints that transition between being unsatisfied and satisfied, then the update time becomes longer. (In the graphical domain this typically corresponds to one part colliding with another, or moving away from another. See [6].)

Another candidate algorithm uses Fourier elimination to compile constraint-free, straight line code that solves simultaneous equalities and inequalities [15]. The resulting code is faster than using QOCA or Cassowary, and it also has uniform running time. However, the algorithm is batch, making it unsuitable for rapidly changing collections of constraints.

To be used in the Ultraviolet framework, a solver for simultaneous inequalities must be able to process the constraints level by level, and after processing each level provide bounds information for variables shared by other regions. Unfortunately neither QOCA nor Cassowary is well-suited for this — these facilities could be provided, but at a higher cost than simply running the algorithm on the entire collection of constraints. Processing the constraints level-by-level requires invoking the solver n times, where n is the number of levels, rather than just once. More significantly, providing bounds information on a level-by-level basis appears to require two optimization steps per variable for each level (to extract the upper and lower bound), which would slow down its operation considerably. However, it appears that the Fourier-based compiler *can* be easily adapted both to process the constraints level-by-level and to extract bounds information.

In view of this, the most direct evolutionary step for Ultraviolet would be to replace Deep Purple with the Fourier elimination algorithm. The result would be a system that solves simultaneous linear equalities and inequalities, along with arbitrary constraints using local propagation. It would be a batch system that produces very efficient, straight-line, constraint-free code. It would thus be useful when the set of constraints is fixed. An example application would be a constraint-based authoring environment for producing Java applets, where the behavior of the applet is partially specified using constraints. After the applet was written and tested in the authoring environment, Ultraviolet would be used to produce Java code that can be shipped over the net and run on a remote machine, without requiring a runtime constraint solver on the remote machine. As a second example, when building an embedded real-time engine controller, predictable performance is needed: compiled code can provide that, but calls to runtime constraint solvers generally cannot. Finally, in developing a product, a company might use constraints in developing the application and not want to ship a proprietary constraint solving package with that application.

For applications in which constraints are added and deleted, an incremental algorithm such as QOCA or Cassowary is needed. In this case an Ultraviolet-like framework could be used to handle non-numeric constraints as well as numeric ones. Given the difficulties in extracting bounds information, *all* of the linear numeric constraints should be collapsed into one region and given to QOCA or Cassowary; the only constraints in the local propagation region would be non-numeric ones. We are currently designing just such a hybrid algorithm.

Acknowledgments

This project has been funded in part by Object Technology International and in part by the National Science Foundation under Grants IRI-9302249 and CCR-9402551. Thanks to Richard Anderson for his help with this project.

References

1. Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 129–136, Seattle, November 1996.
2. Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. The Indigo algorithm. Technical Report 96-05-01, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, July 1996.
3. Alan Borning and Bjorn Freeman-Benson. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 624–628, Cassis, France, September 1995.
4. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.
5. Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of ACM MULTIMEDIA'97*, November 1997.
6. Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, October 1997.
7. Jennifer Burg, Peter J. Stuckey, Jason C.H. Tai, and Roland H.C. Yap. Linear equation solving for constraint logic programming. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 33–47, Tokyo, June 1995. MIT Press.
8. S.S. Chok and K. Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *IEEE Symposium on Visual Languages*, pages 242–250, 1995.
9. Ernest Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, July 1987.
10. Bjorn Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.
11. Michel Gangnet and Burton Rosenberg. Constraint programming and graph algorithms. In *Second International Symposium on Artificial Intelligence and Mathematics*, January 1992.
12. Michael Gleicher. A graphics toolkit based on differential constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 109–120, Atlanta, Georgia, November 1993.
13. Michael Gleicher. *A Differential Approach to Constraint Satisfaction.* PhD thesis, School of Computer Science, Carnegie-Mellon University, 1994.
14. Michael Gleicher. Practical issues in programming constraints. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 407–426. MIT Press, 1995.
15. Warwick Harvey, Peter Stuckey, and Alan Borning. Compiling constraint solving using projection. In *Proceedings of the 1997 Conference on Principles and Practice of Constraint Programming (CP97)*, pages 491–505, October 1997.
16. W. He and K. Marriott. Constrained graph layout. In *Graph Drawing '96*, volume 1190 of *LNCS*, pages 217–232. Springer-Verlag, 1996.

17. Richard Helm, Tien Huynh, Catherine Lassez, and Kim Marriott. A linear constraint technology for interactive graphic systems. In *Graphics Interface '92*, pages 301–309, 1992.
18. Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, DEC Systems Research Center, Palo Alto, CA, 1994.
19. Hiroshi Hosobe, Satoshi Matsuoka, and Akinori Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, Boston, August 1996.
20. Hiroshi Hosobe, Ken Miyashita, Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, pages 51–62. Springer-Verlag LLNCS 874, 1994.
21. Scott Hudson and Ian Smith. SubArctic UI toolkit user's manual. Technical report, College of Computing, Georgia Institute of Technology, 1996.
22. Scott Hudson and Ian Smith. Ultra-lightweight constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 147–155, Seattle, November 1996.
23. Eero Hyvönen. Constraint reasoning based on interval arithmetic: The tolerance propagation approach. *Artificial Intelligence*, 58(1–3):71–112, December 1992.
24. Eero Hyvönen. Evaluation of cascaded interval function constraints. In *Proceedings of the International Workshop on Constraint-Based Reasoning (CONSTRAINT-95)*, Melbourne Beach, Florida, April 1995.
25. Eero Hyvönen, Stefano De Pascale, and Aarno Lehtola. Interval constraint programming in C++. In Brian Mayoh, Enn Tyugu, and Jaan Penjam, editors, *Constraint Programming*, pages 350–366. Springer-Verlag, 1994. NATO Advanced Science Institute Series, Series F: Computer and System Sciences, Vol. 131.
26. Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.
27. Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.
28. Greg Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH '85 Conference Proceedings*, pages 235–243, San Francisco, July 1985. ACM.
29. Michael Sannella. SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 1994 ACM Symposium on User Interface Software and Technology*, pages 137–146, 1994.
30. Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.
31. Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, Department of Electrical Engineering, MIT, January 1963.
32. Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.