Analyzing and Debugging Hierarchies of Multi-Way Local Propagation Constraints

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

April 1, 1994

Abstract

Multi-way local propagation constraints are a powerful and flexible tool for implementing applications such as graphical user interfaces. We have built constraint solvers that maintain sets of preferential multi-way constraints, and integrated them into user interface development environments. These solvers are based on the formal theory of constraint hierarchies, leaving weaker constraints unsatisfied in order to solve stronger constraints if all of the constraints cannot be satisfied.

Our experience has indicated that large constraint networks can be difficult to construct and understand. To investigate this problem, we have developed a system for interactively constructing constraint-based user interfaces, integrated with tools for displaying and analyzing constraint networks. This paper describes the debugging facilities of this system, and presents a new algorithm for enumerating all of the ways that the solver could maintain a set of constraints.

1 Introduction

A multi-way local propagation constraint is represented by a set of *method* procedures that read the values of some of the constrained variables, and calculate values for the remaining constrained variables that satisfy the constraint. A set of such constraints can be maintained by a constraint solver that chooses one method for each constraint so that no variable is set by more than one selected method (i.e., there no *method conflicts*). If there are no cycles in the selected methods, the solver can order them and execute them to satisfy all of the constraints. For example, given the constraint $A + B = C$ (represented by three methods $C \leftarrow A + B$, $A \leftarrow C - B$, and $B \leftarrow C - A$) and the constraint $C + D = E$ (represented by three similar methods), the two constraints could be satisfied by executing the methods $C \leftarrow A + B$ and $E \leftarrow C + D$ in order.

For a given set of constraints, it may not be possible to choose methods for all constraints so there are no method conflicts, or there may be multiple ways to select methods. The theory of constraint hierarchies [1] offers a way to control the behavior of a constraint solver in these situations. Given a constraint hierarchy, a set of constraints where each constraint has an associated *strength*, a constraint solver can leave weaker constraints unsatisfied in order to solve stronger constraints. Research on constraint hierarchies has produced several formal definitions for the "best" solution to a constraint hierarchy that are useful in different applications.

The DeltaBlue and SkyBlue incremental constraint solvers can be used to maintain hierarchies of multi-way local propagation constraints in applications such as user interfaces. Both of these solvers select constraint methods to construct a *method graph* (or *mgraph*) with no method conflicts where stronger constraints are enforced (have a selected method) in favor of weaker constraints. More formally, they construct a locally-graph-better (or LGB) method graph, where a method graph MG is an LGB method graph if it has no method conflicts and for each unenforced constraint C in MG there exists no conflict-free method graph for the same constraints where C is enforced and all of the enforced constraints of MG with the same or stronger strength as C are enforced.

DeltaBlue was the basis of the ThingLab II interactive user interface development environment [5, 10]. SkyBlue is more general successor to DeltaBlue that satisfies cycles of methods by calling external solvers and supports multi-output methods (methods that set multiple output variables) [7, 8]. The Multi-Garnet package [9] uses the SkyBlue solver to add support for multi-way constraints and constraint hierarchies to the Garnet user interface toolkit [6].

As constraint solvers have been applied to larger problems it has become clear that there is a need for constraint network debugging tools. In order to debug a constraint network, the programmer needs tools to examine the constraint network, determine why a given solution is produced, and change the network to produce the desired solution. We have created a system for interactively constructing graphical user interfaces based on constraints (maintained by SkyBlue), and debugging the constraint networks created. The remainder of this paper describes the debugging facilities of this system and presents a new algorithm for generating all LGB method graphs for a set of constraints that promises to be more efficient and useful for debugging common constraint networks.

2 Debugging Constraint Networks

Figure 1 shows two views of a simple user interface constructed using our system. In Figure 1a, the two horizontal lines are lined up. In Figure 1b, the mouse has moved the left endpoint of the bottom line. Constraints keep the width of the line constant, so the right endpoint is moved by the same amount.

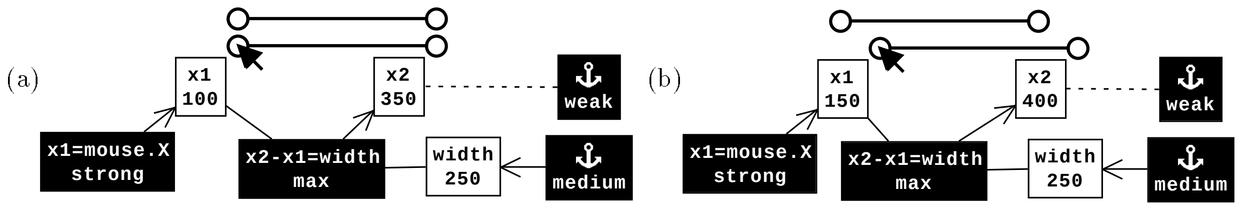

Figure 1: Moving the left endpoint of a constant-width horizontal line.

This figure also displays the constraints relating the variables $x1$ and $x2$, the X-coordinates of the two ends of the bottom horizontal line, to the variable *width*, the difference between the two X-coordinates. The constraint $x1 = mouse.X$ sets $x1$ to the X-coordinate of the mouse position. The *medium* stay constraint (displayed with an anchor symbol) on *width* prevents the solver from changing this variable. The *weak* stay constraint on $x2$ is not enforced, since the solver cannot satisfy this constraint without revoking a stronger constraint. As the mouse is moved, the width of the line is kept constant.

The constraint diagrams present information about the constraints (black boxes) and variables (white boxes) including names, constraint strengths[1] and variable values. The connection between the graphic objects and the variables specifying their positions (such as $x1$) is shown by positioning the variables next to their graphics (though these variable boxes can be moved by the user, if the display gets too complicated). These diagrams also show how each variable value is calculated: the arrows indicate the variables currently determined by the selected method of each constraint. If the constraint is not enforced by any method, the lines to its variables are dashed (i.e., the stay constraint on $x2$). It is possible to explain the derivation of a variable value by examining all of constraints and variables *upstream* of the given variable.

This system can be used to construct complex constraint graphs, and experiment with the behavior of the user interface as constraints are added and removed. We have developed a set of debugging tools that present additional information such as disjoint subgraphs, directed cycles, or directed paths between two variables. A more sophisticated tool analyzes the constraint network to determine why a particular constraint cannot be satisfied, identifying those stronger constraints that prevent the given constraint from being enforced. Similar tools have been developed for the QOCA toolkit [2] and the Geometric Constraint Engine [4]. The following section describes a tool for examining the different possible LGB mgraphs that the solver may

[1]The examples in this paper will use the strengths *max*, *strong*, *medium*, and *weak*, in order from strongest to weakest.

produce, and presents a new algorithm for generating these mgraphs. Research continues on developing new debugging tools, improving the facilities for invoking them, and presenting the results of their analyses.

3 Examining Multiple LGB Method Graphs

When debugging a constraint network, the programmer may want to know whether the constraints specify a unique solution, or whether the solver might produce different solutions at different times. Some constraint solvers can produce different possible solutions for a set of constraints, such as the CLP($\mathcal{R}$) system that generates symbolic expressions representing sets of multiple solutions, and produces alternate solutions upon backtracking [3]. Examining the different solutions can help the programmer understand the constraint network, and determine what constraints should be added to control the solver.

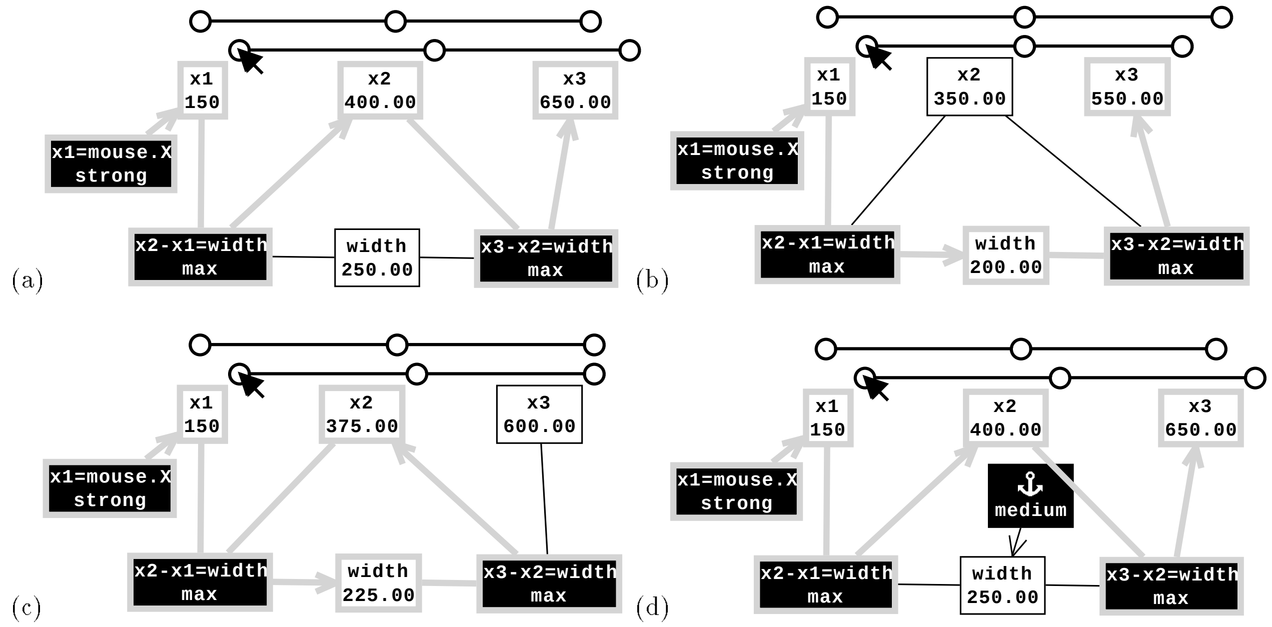

Figure 2: Moving the left endpoint of a horizontal line with a midpoint. The top line shows the initial positions of the three points. Constraints and variables downstream of the mouse constraint are highlighted.

Given a hierarchy of multi-way local propagation constraints, there may be more than one possible LGB method graph that the solver could use to maintain the constraints. For example, consider the constraint network shown in Figure 2. In this situation, there are three ways for the solver to maintain the constraints: by keeping the width variable constant and moving the line (2a), keeping $x2$ constant and moving the two endpoints inward (2b), or keeping $x3$ constant and solving the cycle of linear constraints to position $x2$ between $x1$ and $x3$ (2c). The solver can be forced to choose one of these behaviors by adding *stay* constraints to variables that the user would prefer stay constant (2d). Different strength stay constraints can be used to specify relative preferences for which variables should be constant.

In this example, it is easy to manually generate the possible LGB method graphs. This is much more difficult for large constraint networks: it may not be clear whether there are *any* alternate LGB method graphs. We have developed an algorithm that enumerates all possible LGB method graphs that SkyBlue could produce for a set of constraints. The different method graphs in Figure 2 were generated in this way. Work continues on developing better ways of examining a set of LGB method graphs, such as automatically partitioning them into subclasses depending on which variables are constant. Given two method graphs, it may not be obvious how they differ. Tools have been created for comparing two or more method graphs, highlighting the similarities and differences.

4 An Algorithm for Generating All LGB Method Graphs

We have developed an algorithm for enumerating the possible LGB method graphs for a set of constraints. This algorithm systematically calls the SkyBlue solver to increase the strength of unenforced constraints, searching for alternate method graphs where these constraints are enforced. SkyBlue incrementally updates the current LGB method graph as a constraint is added, removed, or has its strength changed, so it is practical to change constraint strengths repeatedly. The following subsections present this algorithm in stages. First, we present an algorithm for generating all sets of constraints that can be simultaneously enforced in an LGB mgraph. Then, this algorithm is extended to generate all LGB mgraphs.

4.1 LGB Enforced Sets

The *enforced set* (or *E-set*) of an mgraph is the set of constraints that are enforced in the mgraph. The E-set of an LGB mgraph are known as an LGB E-set. For example, Figure 3 shows the two possible LGB mgraphs for the three constraints. These mgraphs have E-sets of $\{C1, C2\}$ and $\{C2, C3\}$ respectively. Sometimes it is useful to speak of the E-set for the constraints with a particular strength. For example, Figure 3a has a *strong* E-set of $\{C2\}$, and a *weak* E-set of $\{C1\}$.

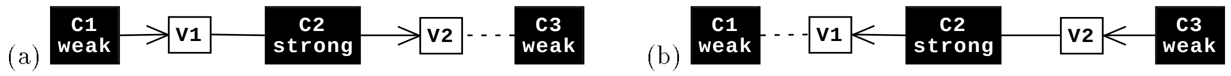

Figure 3: Two LGB mgraphs with different LGB E-sets.

Note that no LGB E-set for a set of constraints can be a proper subset of another LGB E-set. Suppose that E_1 and E_2 are the E-sets for two LGB mgraphs M_1 and M_2 for the same set of constraints. If E_1 were a proper subset of E_2, this would imply that every constraint enforced in M_1 is enforced in M_2, and M_2 contains at least one enforced constraint that is unenforced in M_1, hence M_1 would not be an LGB mgraph.

4.2 Pinning Constraints

Consider an LGB mgraph for a set of constraints. If all of the constraints are enforced, then there is only one possible LGB E-set for these constraints. If some of the constraints are unenforced, then there may be other LGB mgraphs for the constraints where some of the currently-unenforced constraints are enforced and other currently-enforced constraints are unenforced. The question is how to generate them.

Suppose that we start with the LGB mgraph of Figure 3a. All of the *strong* constraints are enforced, so all LGB mgraphs for these constraints will have a strong E-set of $\{C2\}$. Consider the unenforced *weak* constraint $C3$. Suppose that we changed the strength of $C3$ to be slightly stronger than *weak*. In this case, $C3$ would be enforced and $C1$ would be revoked, leading to the mgraph in Figure 4 (the only LGB mgraph for the modified constraints). This mgraph has an E-set of $\{C2, C3\}$, the same as Figure 3b.

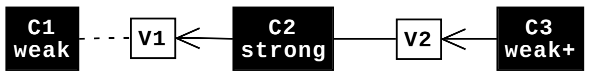

Figure 4: Increasing the strength of $C3$ to produce a different E-set.

For each constraint strength *str*, define the *pin-strength* of *str* as another strength that is slightly stronger than *str*, and weaker than the next stronger constraint strength. The act of increasing the strength of a constraint to its pin-strength (i.e., the pin-strength between its normal strength and the next constraint strength) is called "pinning" the constraint. Pinning different constraints can produce LGB mgraphs with different E-sets, as in Figure 4.

An important fact about pinning is that, no matter what combination of constraints are pinned, the selected methods in the resulting LGB mgraph will specify an LGB mgraph for the original (unpinned) constraints. The algorithm described in the next section systematically pins unenforced constraints to generate different LGB mgraphs for the original constraints, and collects their E-sets.

4.3 Generating LGB E-Sets

Figure 5 presents pseudocode that generates all of the LGB E-sets for a set of constraints. `get_esets` simply initializes global variables containing a list of the constraints we are interested in (`*cns*`), a list of the collected E-sets (`*esets*`), and a procedure to be called to save each E-set (`*save_proc*`).[2] In this case `*save_proc*` is set to the procedure `save_eset`, which adds the E-set for the current mgraph to `*esets*` if it isn't there already. After setting the global variables, `get_esets` calls `pin_cns`, which pins different combinations of constraints, calling `*save_proc*` to process each of the resulting LGB mgraphs.

```
get_esets(cns)
   *cns* := cns
   *esets* := {}
   *save_proc* := save_eset
   pin_cns({}, {}, {}, cns)
   return *esets*

save_eset()
   eset := collect list of all enforced constraints in *cns*
   add eset to *esets* if it is not already there

pin_cns(pinned, unpinned, cns, weaker_cns)
   If cns contains any unenforced constraints then
      cn := choose any unenforced cn in cns
       ;; generate esets with cn unpinned
      pin_cns(pinned, unpinned ∪ {cn}, cns - {cn}, weaker_cns)
       ;; generate esets with cn pinned
      pin(cn)
      If pinned ∪ {cn} are all enforced then
         pin_cns(pinned ∪ {cn}, unpinned, cns - {cn}, weaker_cns)
      unpin(cn)
   Else If weaker_cns is not empty then
       ;; pin all unpinned enforced constraints at current strength
      enforced_unpinned := all enforced constraints in unpinned ∪ cns
      For cn in enforced_unpinned do pin(cn)
       ;; process next weaker cns
      next_strength := strongest strength of constraints in weaker_cns
      next_cns := all constraints in weaker_cns with strength next_strength
      pin_cns({}, {}, next_cns, weaker_cns - next_cns)
       ;; unpin constraints
      For cn in enforced_unpinned do unpin(cn)
   Else
       ;; all cns processed: save current state
      call the procedure *save_proc*

pin(cn)
   cn.original_strength := cn.strength
   change_strength(cn, get_pin_strength(cn.strength))

unpin(cn)
   change_strength(cn, cn.original_strength)
```

Figure 5: Pseudocode to generate all LGB E-sets for `cns`.

[2] All global variables in the pseudocode begin and end with an asterisk. All other variables are local to their procedures.

Most of the work happens in the recursive procedure `pin_cns`. During any call to `pin_cns`, it is processing the set of constraints at a single strength level. The arguments `pinned`, `unpinned` and `cns` are the sets of constraints at the current strength level that have been pinned, left unpinned, and have not been processed. `weaker_cns` contains weaker constraints to be processed later. If `cns` contains any unenforced constraints, one is chosen (`cn`) and `pin_cns` recurses to investigate mgraphs where `cn` is not pinned. When that recursive call returns, `cn` is pinned. If `cn` can be enforced along with all of the other pinned constraints, then `pin_cns` recurses to investigate mgraphs where `cn` is pinned. Finally we unpin `cn`, restoring its original strength.

If there are no unenforced constraints in `cns`, then we have finished processing the constraints at this strength level. Now we are ready to processes the weaker constraints. To ensure that the current strength E-set doesn't change, all of the enforced constraints that haven't been pinned (`enforced_unpinned`) are pinned. Then `pin_cns` recurses, extracting the constraints with the next-weaker strength from `weaker_cns`. Note that when `pin_cns` is initially called from `get_esets` the first three arguments are all empty sets, so `pin_cns` just extracts the strongest constraints from `weaker_cns` and recurses.

When all of the constraints have been processed `*save_proc*` is called to save information about the current LGB mgraph (`get_esets` sets this to `save_eset`, which saves the current LGB E-set).

4.4 Why `get_esets` Generates All E-Sets

Since `get_esets` only modifies the mgraph by pinning constraints, every E-set collected is a correct LGB E-set for the original constraints. To show that `get_esets` is correct, we need to show that every possible E-set is generated. Suppose that this were not true, and there was a set of constraints *cns* with an LGB E-set E that was not generated by `get_esets(`*cns*`)`. Consider the tree of recursive calls to `pin_cns` caused by `get_esets(`*cns*`)`. Figure 6 shows part of such a tree, where first $C1$, and then $C2$, are found to be unenforced, and then either left unpinned or pinned during different recursive calls.

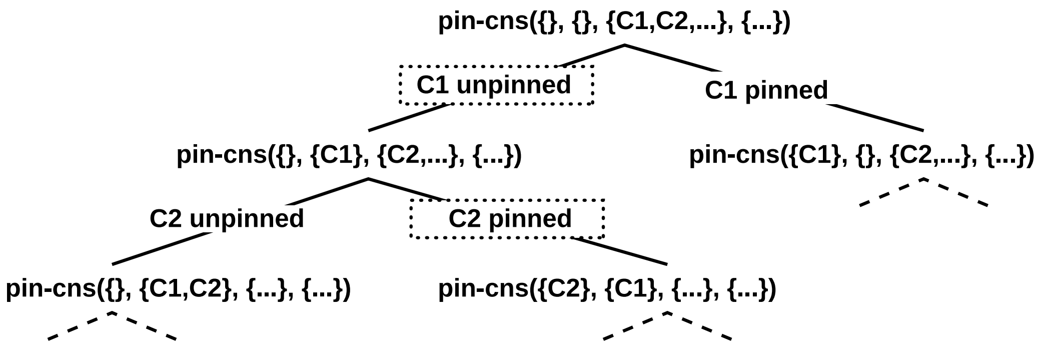
pin-cns({}, {}, {C1,C2,...}, {...})
C1 unpinned
C1 pinned
pin-cns({}, {C1}, {C2,...}, {...})
pin-cns({C1}, {}, {C2,...}, {...})
C2 unpinned
C2 pinned
pin-cns({}, {C1,C2}, {...}, {...})
pin-cns({C2}, {C1}, {...}, {...})

Figure 6: Partial tree of recursive calls to `pin_cns`.

Consider tracing down this tree, following each branch that pins a constraint in E, and each branch that leaves unpinned a constraint that is not in E. For example, if E contained $C2$ and did not contain $C1$, one would follow the branches with boxed labels in Figure 6. Note that every time a constraint in E is pinned it *must* be possible to enforce it along with the other pinned constraints, since E is the E-set for an LGB mgraph, so all of the constraints in E must be simultaneously enforcible. Eventually, you will reach a leaf of the tree, and `*save_proc*` will be called to process the current mgraph.

We claim that the E-set of this leaf mgraph is exactly E. First, all of the constraints in E that were found unenforced in `cns` and pinned are enforced. Consider some other constraint *cn* that is in E but was not found unenforced in `cns`. It must not have been removed from `cns`, since the only way constraints are removed from `cns` is when they are considered for pinning, and if this had happened then *cn* would have been explicitly pinned. Since it wasn't removed from `cns`, then it must have been enforced when that strength level was processed, and hence it was pinned when going to the next strength level. Therefore, it must be enforced in the final mgraph, and all of the constraints in E must be enforced. Finally, consider some constraint cn'

that is not in E. If it was enforced, then there would be an LGB mgraph where all of the constraints in E plus another constraint cn' are enforced, in which case E would not be the E-set of an LGB mgraph. Thus, we have shown that exactly those constraints in E are enforced in the leaf mgraph.

4.5 Generating Results Multiple Times

The procedure `save_eset` is written to add the current E-set to the list `*esets*` only if it is not there already. This is necessary because `get_esets` may generate the same E-set multiple times if constraints have multi-output methods. For example, suppose we call `get_esets` on the three *strong* constraints $C1$, $C2$, and $C3$, whose current mgraph is shown in Figure 7a. The clock diagram indicates that $C1$ has a single method which outputs to both $V1$ and $V2$ (this diagram is not shown for constraints with a single-output method outputting to each of their variables). If we pin $C2$ and not $C3$, we produce the mgraph of Figure 7b, and collect its E-set. On backtracking, if we leave $C2$ unpinned, and pin $C3$, we will produce Figure 7c, which has the same E-set.

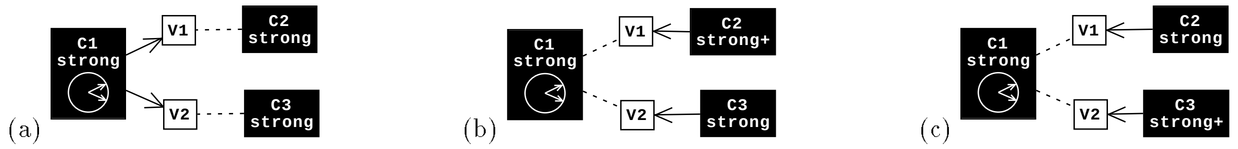

Figure 7: Starting with mgraph (a), `get_esets` may collect the same E-set multiple times (b,c).

4.6 Collecting Some LGB Method Graphs by Adding Stay Constraints

It would be possible to modify `save_eset` to collect the enforced constraints along with their current selected methods when it is called within `get_esets`. If the given set of constraints had exactly one LGB mgraph for each LGB E-set, this would collect all of the LGB mgraphs. However, if there are multiple LGB mgraphs that have the same E-set (Figure 8), there is no guarantee that they would all be generated by `get_esets`.

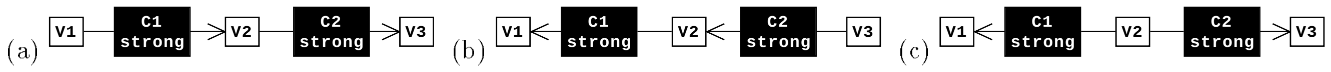

Figure 8: Three possible LGB mgraphs for $\{C1, C2\}$.

One thing that distinguishes different LGB mgraphs with the same E-set is the sets of variables that are determined and undetermined. This observation can be used to generate these different LGB mgraphs: Given a set of constraints *cns*, let *v-weak* be a strength weaker than any of these constraints. For each of the variables that can be determined by any of the constraints' methods (the *potential outputs* of the constraints), add a new stay constraint with strength *v-weak*. Consider an LGB mgraph for this extended set of constraints, *cns'*. The selected methods for *cns* in the extended mgraph define an LGB mgraph for *cns* alone, since none of the *v-weak* stay constraints can effect which stronger constraints are enforced, but they can effect the selected methods used to enforce stronger constraints. Calling `pin_cns(`*cns'*`)` will pin all of the constraints including the *v-weak* stay constraints, generating different LGB mgraphs for *cns*. For example, Figure 9 shows how extra *v-weak* stay constraints added to the constraints from Figure 8 can be pinned to generate the mgraphs in Figure 8a and 8c.

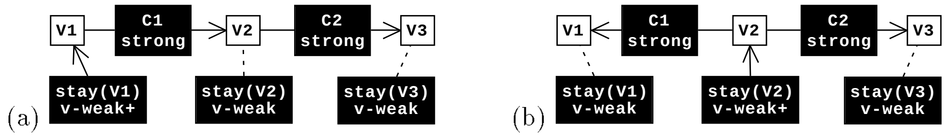

Figure 9: Pinning extra stay constraints to generate different LGB mgraphs for $\{C1, C2\}$.

Figure 10 presents pseudocode that creates the extra variable stay constraints and passes all of these constraints to `pin_cns`, which will generate different LGB mgraphs for `cns`. Note that `*save_proc*` is set to the procedure `save_mgraph`, so it will be called to save the current mgraph (including selected methods) within `pin_cns`. `*cns*` does not include the extra variable stay constraints, since we are only concerned with collecting the method graphs for the original constraints.

```
get_some_lgb_mgraphs(cns)
   *cns* := cns
   *mgraphs* := {}
   *save_proc* := save_mgraph
    ;; add stays to output variables
   var_stay_strength := any strength weaker than all of the constraints in cns
   potential_outputs := a list of all potential output variables for cns
   output_var_stays := a stay constraint with strength var_stay_strength
                       for each var in potential_outputs
   For cn in output_var_stays do add_constraint(cn)
    ;; generate mgraphs for constraints, including extra stays
   pin_cns({}, {}, {}, cns ∪ output_var_stays)
    ;; remove added stays
   For cn in output_var_stays do remove_constraint(cn)
   return *mgraphs*

save_mgraph()
   mgraph := For cn in *cns* collect cn and its current selected mt
   add mgraph to *mgraphs* if it is not already there
```

Figure 10: Pseudocode to generate some LGB mgraphs for `cns`.

4.7 Collecting All LGB Method Graphs by Adding Method Variables

There are two situations where `get_some_lgb_mgraphs` may not generate all possible LGB mgraphs for a set of constraints: (1) There are directed cycles of methods. The constraints $\{C1, C2\}$ in Figure 11a have two LGB mgraphs, one with a directed cycle in each direction. Pinning extra stay constraints on the variables will not choose one mgraph over the other. (2) There are constraints with "subset methods," where the outputs of one constraint method are a subset of the outputs of another method for the same constraint. This is rare, but it is not prohibited by the definition of multi-way local propagation constraints. For example, constraint $C3$ in Figure 11b has one method that outputs to $V7$ and $V8$, and another method that outputs to $V8$. If the constraint solver always chooses the second method, `get_some_lgb_mgraphs` will never generate an mgraph containing the first method.

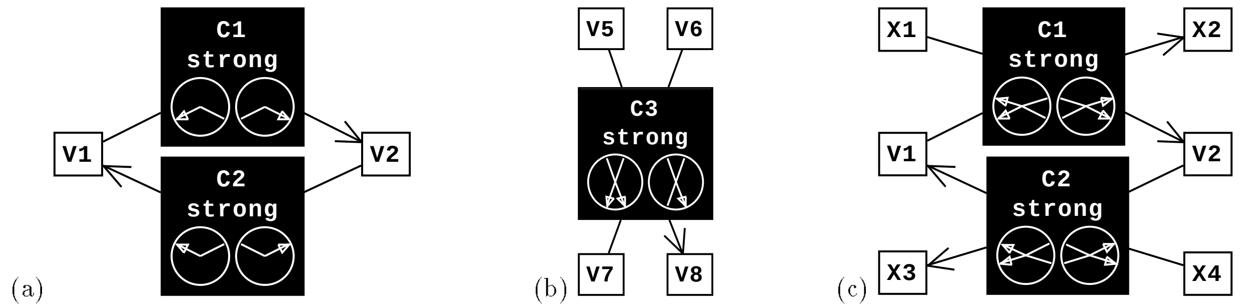

Figure 11: (a) A method graph with a directed method cycle. (b) A constraint with subset methods. (c) Adding extra variables to (a). Method diagrams are shown in (a) for comparison with (c).

Given cycles or subset methods, it is possible to generate all possible mgraphs using the pseudocode of Figure 12. This code modifies every constraint that has more than one method, creating an extra variable for each constraint method, and adding it as an output to all of the *other* methods of the constraint. Applying this to Figure 11a produces Figure 11c, where the new variable $X1$ is only set when $C1$ is enforced with a method other than its second method (setting $V2$ and $X2$). When the modified constraints are passed to `get_some_lgb_mgraphs`, and *v-weak* stays are added to these extra variables, pinning these extra stays will try all of the methods of each constraint, if they are allowed in an LGB mgraph. Note that constraints with only a single method do not need to have any extra variables added, since such a constraint's single method must be used whenever the constraint is enforced.

```
get_all_lgb_mgraphs(cns)
    ;; add extra variables to methods
   For all constraints cn in cns with more than one method do
      remove_constraint(cn)
      For mt in cn.methods do
         v := create a new variable
         add v to cn.variables
         add v to the outputs of all of cn's methods except mt
      add_constraint(cn)
    ;; add extra stays to variables, and generate mgraphs
   mgraphs := get_some_lgb_mgraphs(cns)
    ;; remove extra variables from constraints and methods
   For all constraints cn in cns with more than one method do
      remove_constraint(cn)
      restore cn.variables
      restore outputs for all of cn's methods
      add_constraint(cn)
   return mgraphs
```

Figure 12: Pseudocode to generate all LGB mgraphs for `cns`.

The pseudocode removes all of the constraints before adding the additional variables to the methods, and then re-adds the constraints. Likewise, the constraints are removed before removing these additional variables. If the constraint solver had an entry for modifying methods, this would not be necessary.

4.8 Evaluating the Algorithms

We are currently comparing the performance of `get_all_lgb_mgraphs` to alternate algorithms for generating LGB mgraphs. An earlier algorithm enumerated all possible combinations of selected methods without method conflicts, collecting all mgraphs that were LGB. This worked well for small networks, but was much too slow for large networks (taking time exponential in the number of constraints). Testing shows that `get_all_lgb_mgraphs` is much faster than the earlier algorithm for many sets of constraints encountered in actual practice (2 seconds versus 22 minutes for one set of 17 constraints), but we have been able to construct constraint networks where it is significantly slower than the earlier algorithm. `get_all_lgb_mgraphs` appears to be most efficient when there are only a few possible LGB mgraphs. In the absence of a simple way to predict which algorithm is faster, it might be reasonable to run both algorithms in parallel.

The different algorithms described in this paper may be useful at different points during debugging. `get_esets` can be used to determine whether a given constraint is always enforced, or never enforced. If there are no subset methods and the programmer doesn't care about the directions of cycles, `get_some_lgb_mgraphs` can be called instead of `get_all_lgb_mgraphs`.

These algorithms call the SkyBlue constraint solver to manipulate the constraints. Therefore, any future performance improvements to SkyBlue (or other algorithms that maintain LGB mgraphs) will directly improve the performance of these algorithms.

5 Conclusions and Future Work

We have described some of the debugging tools included within our system for interactively constructing constraint-based user interfaces, and presented a new algorithm for generating all of the LGB method graphs for a set of constraints. This algorithm is the basis for a powerful debugging tool that allows the programmer to explore the different behaviors that can be produced by a set of constraints.

In the future we want to continue developing new debugging tools, improving the facilities for invoking them and presenting the results of their analyses. We also want to conduct user testing, to determine which debugging tools are particularly helpful to programmers when constructing large constraint networks.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant IRI-9102938.

References

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

[2] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-oriented Graphics*, Champery, Switzerland, October 1992. Also will be published in *Advances in Object-Oriented Graphics II*, Springer-Verlag, 1993.

[3] 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.

[4] Walid T. Keirouz, Glenn A. Kramer, and Jahir Pabon. Exploiting constraint dependency information for debugging and explanation. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 156–165, April 1993.

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

[6] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[7] Michael Sannella. The SkyBlue constraint solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993.

[8] Michael Sannella. The SkyBlue constraint solver and its applications. In Saraswat and van Hentenryck, editors, *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming.* MIT Press, 1994. To appear.

[9] Michael Sannella and Alan Borning. Multi-Garnet: Integrating multi-way constraints with Garnet. Technical Report 92-07-01, Department of Computer Science and Engineering, University of Washington, September 1992.

[10] 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.