# Locally Simultaneous Constraint Satisfaction

Hiroshi Hosobe,[1*] Ken Miyashita,[1] Shin Takahashi,[1]
Satoshi Matsuoka,[2] and Akinori Yonezawa[1]

[1] Department of Information Science, University of Tokyo
[2] Department of Mathematical Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

**Abstract.** Local propagation is often used in graphical user interfaces to solve constraint systems that describe structures and layouts of figures. However, algorithms based on local propagation cannot solve simultaneous constraint systems because local propagation must solve constraints individually. We propose the '*DETAIL*' algorithm, which efficiently solves systems of constraints with strengths, even if they must be solved simultaneously, by 'dividing' them as much as possible. In addition to multi-way constraints, it handles various other types of constraints, for example, constraints solved with the least squares method. Furthermore, it unifies the treatment of different types of constraints in a single system. We implemented a prototype constraint solver based on this algorithm, and evaluated its performance.

## 1 Introduction

*Local propagation* is an efficient constraint satisfaction algorithm that takes advantage of potential locality of constraint systems. It is often used in graphical user interfaces (GUIs) to solve constraint systems that describe structures and layouts of figures.

Recent constraint solvers based on local propagation handle *multi-way constraints* [4]. A multi-way constraint can be solved for any one of its variables. For example, the constraint $x = y + z$ is multi-way because it can be transformed into $x \leftarrow y + z$, $y \leftarrow x - z$, and $z \leftarrow x - y$. Local propagation satisfies systems of multi-way constraints by solving each constraint at most once in some order. For example, consider a constraint system with the constraints $v = w \times x$, $w = y$, and $x = y + z$. Figure 1a shows a *constraint graph* representing this system, where circles and squares represent variables and constraints respectively. This system can be satisfied by solving $x \leftarrow y + z$, $w \leftarrow y$, and $v \leftarrow w \times x$ in this order. This case is illustrated by the *correct solution graph* in Fig. 1b, where arrows from constraints point to variables to which the constraints output values. A solution graph is a constraint graph that dictates how each constraint will be solved, and a correct solution graph satisfies the following two properties: (1) the value of each variable must be determined by at most one constraint, that is, the graph should have no *conflicts*, and (2) all the constraints must be partially ordered, that is, the graph must have no *cycles*.

* E-mail: detail@is.s.u-tokyo.ac.jp

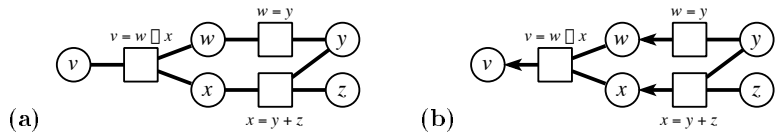


**Fig. 1.** (a) A constraint graph and (b) its correct solution graph

Multi-way constraints embody a problem that correct solution graphs are not determined uniquely. Borning et al. proposed *constraint hierarchies* to cope with this problem [7]. A constraint hierarchy is a system of constraints with hierarchical *strengths*. If the system is over-constrained, it is solved so that there are as many satisfied strong constraints as possible, which allows programmers to implicitly specify solution graphs. In Fig. 2a, for example, the constraints $x = 1$ and $x = 3$ conflict. However, if $x = 1$ and $x = 3$ are associated with strong and weak respectively, the constraint system is solved by satisfying only $x = 1$ as shown in Fig. 2b. DeltaBlue is the first proposed algorithm that efficiently solves hierarchies of multi-way constraints [2, 5]. It determines output variables of constraints incrementally when a constraint is added or removed, and realizes constraint satisfaction without spoiling the efficiency of local propagation.


**Fig. 2.** (a) A solution graph for an over-constrained system and (b) one for a constraint hierarchy

Local propagation has a serious problem that constraint systems employed in real applications often result in solution graphs with cycles or conflicts. For example, consider a constraint system with the constraints $a - b = l$, $(a + b)/2 = m$, stay($l$), and edit($m$). This system represents a typical situation where the midpoint of two points is moved with a mouse, but its solution graphs contain cycles by necessity, e.g. as illustrated in Fig. 3a. As another example, suppose a constraint hierarchy with the constraints strong $x = 1$ and strong $x = 3$. Even if one wants to apply the least squares method to these constraints and to obtain the solution $x = 2$, the resulting solution graph contains a conflict as shown in Fig. 3b. Generally, in constraint systems that result in solution graphs with cycles or conflicts, constraints need to be solved simultaneously.

We propose the '*DETAIL*' algorithm, which efficiently solves constraint hierarchies, even if constraints must be solved simultaneously, by 'dividing' them as

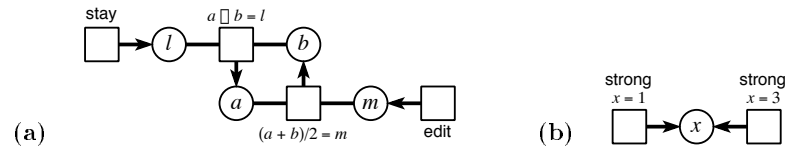


**Fig. 3.** (a) A solution graph with a cycle and (b) one with a conflict

much as possible. This algorithm is efficient enough to be applied to constraint-based GUIs since it incrementally finds parts of constraint systems that must be solved simultaneously. In addition to multi-way constraints, it handles various other kinds of constraints, for example, constraints solved with the least squares method. Furthermore, it unifies the treatment of different types of constraints in a single hierarchy. We implemented a prototype constraint solver based on this algorithm, and evaluated its performance.

## 2 Locally Simultaneous Constraint Satisfaction

In this section, we present an extended theory of constraint hierarchies and the *DETAIL* algorithm.

### 2.1 Constraints

In our extended constraint hierarchy theory, constraints are categorized into *solution types*, which are determined by how the constraints are solved. For example, there is a solution type of constraints that will be ignored if they cannot be solved exactly, as with the DeltaBlue algorithm. Also, there is another solution type of constraints that must be solved even in such a case by minimizing their errors with the least squares method.

All constraints with an equal strength must belong to a single solution type. Intuitively, this requirement is necessary because it is difficult to equally treat constraints of different solution types.

Based on this theory, the *DETAIL* algorithm solves hierarchies of multi-way constraints where all constraints are independent. For example, a hierarchy must not contain the constraints strong $x + y = 1$ and weak $x + y = 1$.

### 2.2 Theory

By extending the theory described in [7], we formulated constraint hierarchies that contain multiple solution types of constraints. A constraint hierarchy $H$ is a pair $(V, C)$, where $V$ is a set of variables that range over some domain $\mathcal{D}$, and $C$ is a set of constraints on variables in $V$. Each constraint is associated with a strength $i$ where $0 \leq i \leq n$. Strength 0 represents the strength of required

constraints, and the larger the number of a strength, the weaker it is. All constraints with an equal strength $i$ are categorized into a solution type $\tau_i$. $C$ is divided into a set of lists $\{C_0, C_1, \ldots, C_n\}$, where $C_i$ contains constraints with strength $i$ in some arbitrary order.

Solutions to a constraint hierarchy are defined as a set of valuations. A valuation $\theta$ is a function that maps variables in $V$ to their values in $\mathcal{D}$. An error function $e_\tau$ returns a non-negative real by evaluating the error for $\theta$ of a constraint $c$ of a solution type $\tau$. The error $e_\tau(c\theta) = 0$ if and only if $c$ is exactly satisfied by $\theta$. The function $E_{\tau_i}$ returns the list of errors of a list of constraints $C_i = [c_1, c_2, \ldots, c_k]$, i.e.,

$$E_{\tau_i}(C_i\theta) = [e_{\tau_i}(c_1\theta), e_{\tau_i}(c_2\theta), \ldots, e_{\tau_i}(c_k\theta)] \ .$$

Each element $e_{\tau_i}(c_i\theta)$ can be weighted by a positive real $w_i$. An error sequence $R(C\theta)$ is the error of $C$ except $C_0$:

$$R(C\theta) = [E_{\tau_1}(C_1\theta), E_{\tau_2}(C_2\theta), \ldots, E_{\tau_n}(C_n\theta)] \ .$$

A combining function $g_{\tau_i}$ combines $E_{\tau_i}(C_i\theta)$ into a value of a domain where elements are comparable. Two combined errors $g_{\tau_i}(E_{\tau_i}(C_i\theta))$ and $g_{\tau_i}(E_{\tau_i}(C_i\varphi))$ are compared by a reflexive and symmetric relation $<>_{g_{\tau_i}}$, and an irreflexive, antisymmetric, and transitive relation $<_{g_{\tau_i}}$. The function $G$ combines an error sequence $R(C\theta)$:

$$G(R(C\theta)) = [g_{\tau_1}(E_{\tau_1}(C_1\theta)), g_{\tau_2}(E_{\tau_2}(C_2\theta)), \ldots, g_{\tau_n}(E_{\tau_n}(C_n\theta))] \ .$$

Two combined error sequences $G(R(C\theta))$ and $G(R(C\varphi))$ are compared by a lexicographic ordering $<_G$:

$$\begin{aligned} G(R(C\theta)) <_G G(R(C\varphi)) \equiv\ & \exists k \in \{1, 2, \ldots, n\}. \\ & \forall i \in \{1, 2, \ldots, k-1\}.\ g_{\tau_i}(E_{\tau_i}(C_i\theta)) <>_{g_{\tau_i}} g_{\tau_i}(E_{\tau_i}(C_i\varphi)) \ \wedge \\ & g_{\tau_k}(E_{\tau_k}(C_k\theta)) <_{g_{\tau_k}} g_{\tau_k}(E_{\tau_k}(C_k\varphi)) \ . \end{aligned}$$

We say that $\theta$ is *better* than $\varphi$ if and only if $G(R(C\theta)) <_G G(R(C\varphi))$.

The set $S$ of solutions to $H$ is defined as follows:

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in C_0.\ e_{\tau_0}(c\theta) = 0\} \\ S &= \{\varphi \in S_0 \mid \forall \theta \in S_0. \neg(G(R(C\theta)) <_G G(R(C\varphi)))\} \ . \end{aligned}$$

The main difference from the original formulation in [7] is existence of solution types. In [7], all constraints in a constraint hierarchy are categorized into some single solution type, and therefore, for each strength $i$, $e_{\tau_i}$ and $g_{\tau_i}$ are some $e$ and $g$ respectively. Since errors of constraints with different strengths are never compared directly, we can safely assign various solution types to each strength.

Two error functions are presented in [7]: Given a constraint $c$ and a valuation $\theta$, the *metric* error function returns $c$'s metric, e.g. for the constraint $x = y$, the distance between $x$ and $y$. Also, the *predicate* error function returns 0 if $c$ is exactly satisfied for $\theta$, and 1 otherwise.

Also in [7], several combining functions and associated relations are provided. Since it does not introduce multiple solution types in a constraint hierarchy, an instance of $<_G$ is determined by single instances of $e$ and $g$. For an instance of $<_G$ called *least-squares-better*, given lists of errors $\mathbf{v} = [v_1, v_2, \ldots, v_k]$ obtained with the metric error function, $g(\mathbf{v}) = \sum_{i=1}^{k} w_i v_i^2$, $<_g$ is $<$ and $<>_g$ is $=$ for reals. For instances of $<_G$ called *locally-better*, given $\mathbf{v} = [v_1, v_2, \ldots, v_k]$ and $\mathbf{u} = [u_1, u_2, \ldots, u_k]$, $g(\mathbf{v}) = \mathbf{v}$ and $<_g$ and $<>_g$ are defined as follows:

$$\mathbf{v} <_g \mathbf{u} \equiv \forall i.\ v_i \leq u_i \ \wedge\ \exists j.\ v_j < u_j$$
$$\mathbf{v} <>_g \mathbf{u} \equiv \forall i.\ v_i = u_i \ .$$

*Locally-predicate-better* is the locally-better using the predicate error function, and *locally-metric-better* is the one employing the metric error function.

In the rest of this paper, we refer to the solution type associated with least-squares-better as $\tau_{\mathrm{LSB}}$ and constraints of $\tau_{\mathrm{LSB}}$ as least-squares-better constraints, and correspondingly locally-predicate-better as $\tau_{\mathrm{LPB}}$ and locally-predicate-better constraints.[3]

### 2.3 Solution Graphs

Local propagation cannot solve conventional solution graphs with cycles or conflicts. To cope with this problem, we propose a new definition of solution graphs. Before presenting it, we define constraint graphs of constraint hierarchies.

**Definition 1 (Constraint graph).** Given a constraint hierarchy $H = (V, C)$, a bipartite graph $B = (V, C, E)$, where $V$ and $C$ are sets of nodes and $E$ is a set of edges, is a constraint graph of $H$ if and only if

$$E = \{(v, c) \in V \times C \mid v \text{ is constrained by } c\} \ .$$

We say that $v$ and $c$ are *adjacent* if and only if $(v, c) \in E$.

We define solution graphs using *constraint cells* to overcome the defects of conventional solution graphs.

**Definition 2 (Constraint cell).** Let $H = (V, C)$ be a constraint hierarchy, and $B = (V, C, E)$ a constraint graph of $H$. For $X \subseteq V$, define $\Gamma$ as follows:

$$\Gamma(X) = \{c \mid (v, c) \in E \ \wedge\ v \in X\} \ .$$

A pair $p = (V_p, C_p)$ is a constraint cell in $B$ if and only if:

1. $V_p \subseteq V$, $C_p = \emptyset$, and $|V_p| = 1$, or
2. $V_p \subseteq V$, $C_p \subseteq C$, the subgraph of $B$ induced by $V_p$ and $C_p$ is connected, and

$$\forall X \subseteq V_p.\ |X| \leq |\Gamma(X) \cap C_p| \ .$$

[3] These names may sound strange because 'better' is associated with $<_G$ (not $<_g$), but we use them to avoid introducing new terminologies.

We say that $p$ is *over-constrained* if and only if $|V_p| < |C_p|$.

Values of variables in a constraint cell are obtained by evaluating constraints in the cell. Because of Definition 2, this is always possible for constraints that we handle. Definition 2 is based on Hall's theorem, known in graph theory, which describes the condition on existence of perfect matchings of bipartite graphs. Intuitively, Definition 2 means that given a constraint cell $p = (V_p, C_p)$, the value of each variable in $V_p$ can be determined by at least one constraint in $C_p$.

**Definition 3 (Solution graph).** Given a constraint graph $B = (V, C, E)$ and a set $P$ of constraint cells in $B$, a quadruple $B_S = (V, C, E, P)$ is a solution graph for $B$ if and only if:

1. each variable in $V$ belongs to only one constraint cell in $P$,
2. each constraint in $C$ belongs to only one constraint cell in $P$, and
3. there are no cyclic dependencies among constraint cells in $P$.

For example, Fig. 4 shows a solution graph equivalent to the one in Fig. 1b, where boxes with round corners illustrate constraint cells.[4]

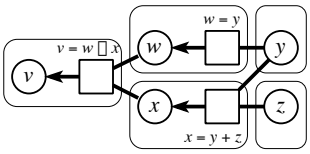


**Fig. 4.** A solution graph with constraint cells

Constraint cells are created so that they contain cycles and conflicts. In addition, over-constrained cells are sometimes merged with other cells to produce a 'better' solution graph, i.e. the corresponding valuation is better, because the new cells may acquire more freedom to determine the values of their variables. For example, consider a constraint hierarchy with the constraints $\alpha$, $\beta$, $\gamma$, $\delta$, $\epsilon$, $\zeta$, $\eta$, and $\theta$. Let $\alpha$ be required $t = 0$, $\beta$ weak $t = u$, $\gamma$ weak $v = 1$, $\delta$ strong $t + v = w$, $\epsilon$ weak $w = x$, $\zeta$ strong $x + y = z$, $\eta$ required $x + 1 = y$, and $\theta$ medium $z = 7$, where strong and medium constraints are locally-predicate-better, and weak constraints are least-squares-better. Figure 5a shows a solution graph of this hierarchy. Satisfying constraints locally in these cells, the corresponding valuation $\Theta$ is obtained as $\{t \mapsto 0,\ u \mapsto 0,\ v \mapsto 1,\ w \mapsto 1,\ x \mapsto 3,\ y \mapsto 4,\ z \mapsto 7\}$, and the combined error sequence is $[g_{\tau_{\mathrm{LPB}}}(E_{\tau_{\mathrm{LPB}}}([\delta, \zeta]\Theta)), g_{\tau_{\mathrm{LPB}}}(E_{\tau_{\mathrm{LPB}}}([\theta]\Theta)), g_{\tau_{\mathrm{LSB}}}(E_{\tau_{\mathrm{LSB}}}([\beta, \gamma, \epsilon]\Theta))]$ $= [[0, 0], [0], 4]$. By contrast, merging the over-constrained cell $W$ and the cell $V$

[4] For readability, we often draw arrowheads in constraint cells although they are not essential.

into the new cell $W'$, we obtain the solution graph in Fig. 5b,[5] and then, the corresponding valuation $\Phi$ is $\{t \mapsto 0,\ u \mapsto 0,\ v \mapsto 2,\ w \mapsto 2,\ x \mapsto 3,\ y \mapsto 4,\ z \mapsto 7\}$, and the combined error sequence is $[[0,0],[0],2]$. This indicates that $\Phi$ is better than $\Theta$.

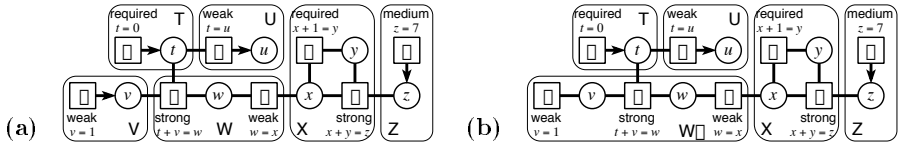


**Fig. 5.** Merging an over-constrained cell with another constraint cell

We define correct solution graphs using *internal strengths* and *walkabout strengths* of constraint cells so that the graphs can produce solutions to constraint hierarchies. Walkabout strengths were first introduced in the DeltaBlue algorithm, but for our purpose, we extend its definition:

**Definition 4 (Internal strength).** The internal strength of a constraint cell $p = (V_p, C_p)$ is (1) weakest if $C_p = \emptyset$, or (2) the weakest among strengths of constraints in $C_p$, otherwise.

**Definition 5 (Walkabout strength).** The walkabout strength of a constraint cell $p$ is the weakest among $p$'s internal strength and walkabout strengths of constraint cells with variables adjacent to the constraints in $p$.

**Definition 6 (Correct solution graph).** A solution graph is correct if and only if:

1. for each constraint cell with multiple constraints, the pair of the set of its variables and the set of its non-weakest constraints does not constitute a constraint cell, i.e. does not satisfy Definition 2, and
2. for each over-constrained cell, its internal strength is weaker than the walkabout strengths of any other constraint cells with the variables adjacent to the constraints in the over-constrained cell.

Intuitively, Condition 1 of Definition 6 makes constraint cells use the weakest constraints to determine the values of their variables, and Condition 2 guarantees that constraints in over-constrained cells cannot override constraints in other cells even if they are merged.

[5] Note that constraint cells are *not* merged simply because they contain constraints of similar solution types or constraints with equal strengths. For example, $W'$ in Fig. 5b contains multiple solution types of constraints with multiple strengths

### 2.4 Algorithm

It is desirable that sizes of constraint cells in correct solution graphs are minimized since local propagation can be efficiently applied to such graphs. The *DETAIL* algorithm creates such solution graphs incrementally when invoked with the following five operations: adding a variable, removing a variable, adding a constraint, removing a constraint, and updating a variable value. The former four operations cause the corresponding solution graph to be modified, and the last operation applies local propagation to the solution graph as described earlier. We call the former *planning* and the latter *execution*.

The algorithm for adding or removing a variable is quite straightforward: to add a variable, *DETAIL* only creates a new constraint cell with the variable, and to remove a variable, it deletes the constraint cell with the variable after verifying that the variable is not adjacent to any constraints. In the rest of this section, we describe the algorithm for adding or removing a constraint to a hierarchy.

**Adding a Constraint.** Initially, there is a correct solution graph whose constraint cells are minimized. When a new constraint is added to this hierarchy, one or more constraints with an equal or weaker strength may be 'victimized,' that is, their associated errors will be increased. In such a case, *DETAIL* reconstructs the solution graph incrementally to keep it correct and its constraint cells minimal by modifying the necessary set of cells.

*DETAIL* treats locally-predicate-better constraints specially by permitting 'equal to' as well as 'weaker than' in Condition 2 of Definition 6, because these constraints can be ignored if they cannot be exactly satisfied, and resulting solution graphs may be solved more efficiently.

Figure 6 shows the algorithm that adds a constraint *con* to a constraint hierarchy, and Fig. 7 describes the algorithm to decompose a constraint cell at lines 12 and 17 in Fig. 6. The former algorithm works as follows: First, it creates a constraint cell with *con* at line 1. Second, it finds the strength of the 'victim' constraint at line 2. Next, it follows the path from *con* to the victim at lines 4–19, reversing the dependency between the cells along the path. After this process, *con* becomes active. Then, it eliminates cycles of constraint cells generated in the previous process at line 20, and updates walkabout strengths correctly at line 21. Finally, it merges over-constrained cells with others at line 23 so that they can minimize the errors of their constraints.

Figure 8 shows an example of the execution of this algorithm. Initially, there is a correct solution graph illustrated in Fig. 8a. When a constraint $\theta$ is added to the constraint hierarchy, this algorithm works as follows:

1. A constraint cell $H$ with $\theta$ is created (Fig. 8b). The strength of the victim is found to be weak.
2. After the variable $z$ is removed from the cell $G$, it is added to $H$ (Fig. 8c).
3. The variable $x$ is deleted from the cell $E$, and is added to $G$ (Fig. 8d). The weak constraint $\epsilon$ in $E$ is found to be the victim.
4. The constraint cells $G$ and $F$ are merged because they form a cycle.

```
1   cl ← a new cell with con;
2   wastr ← the weakest of walkabout strengths of
        cells with variables adjacent to con;
3   str ← con's strength;
4   while str is stronger than wastr do
5     nextcl ← a cell with a variable adjacent to a constraint in cl and
          with walkabout strength wastr;
6     var ← a variable in nextcl that connects to cl;
7     remove var from nextcl;
8     add var to cl;
9     if nextcl is empty then
10      str ← weakest;
11    else if nextcl's internal strength is wastr then
12      cl ← an over-constrained cell generated by decomposing nextcl;
13      str ← cl's internal strength;
14    else
15      bordercon ← a constraint in nextcl and
            adjacent to a variable in a cell with walkabout strength wastr;
16      remove bordercon from nextcl;
17      decompose nextcl;
18      cl ← a new cell with bordercon;
19      str ← cl's internal strength; /* end of while */
20  merge cyclic cells dependent on con;
21  update walkabout strengths of cells dependent on con;
22  if wastr is not weakest and constraints with strength wastr are
        not locally-predicate-better constraints then
23    merge cells that cl depends on and
          that have the same walkabout strength as cl;
```

**Fig. 6.** Adding a constraint *con* to a constraint hierarchy.

```
1   for each variable var in cl do
2     remove var from cl;
3     create a cell with var;
4   for each constraint con stronger than wastr in cl do
5     remove con from cl;
6     var ← a variable initially in cl that forms a cell alone and
          that con depends on;
7     reverse the dependency between con and var;
8   for each constraint con with strength wastr in cl do
9     remove con from cl;
10    if there is a variable initially in cl that forms a cell alone and
          that con depends on then
11      var ← the variable found above;
12      reverse the dependency between con and var;
13    else
14      create a cell with con;
```

**Fig. 7.** Decomposing a constraint cell *cl* with walkabout strength *wastr*

5. Walkabout strengths are updated (Fig. 8e).
6. Since $E$ is over-constrained, it is joined with the constraint cells $D$ and $C$, which have the same walkabout strength `weak` as $E$ (Fig. 8f).

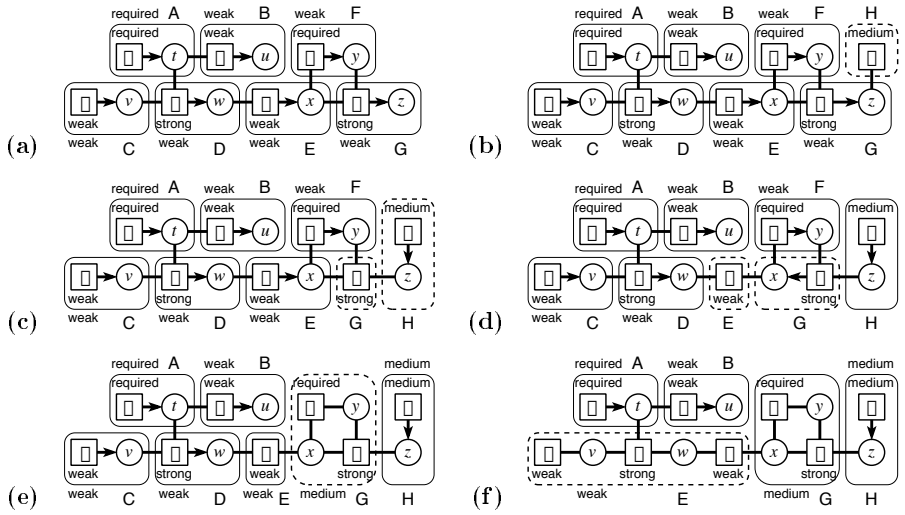


**Fig. 8.** Adding a constraint

It is sometimes necessary to decompose 'large' constraint cells that contain multiple constraints. Figure 7 describes the algorithm that decomposes 'large' cells into 'small' ones. Basically, it matches variables with constraints, employing a perfect matching algorithm for bipartite graphs. In addition, since the weakest constraints sometimes need to remain unsatisfied, it later tries to match the weakest constraints at lines 8–14. Definition 2 of constraint cells guarantees that there are no undetermined variables after decomposing cells with one or more constraints. Even if constraint cells that do not satisfy Condition 3 in Definition 3 or Condition 2 in Definition 6 are generated, they will be merged by the caller algorithm in Fig. 6.

**Removing a Constraint.** Removing a constraint from a constraint hierarchy may cause one or more constraints with an equal or weaker strength to decrease their errors, because it or they may acquire more freedom to determine the value of variables instead of the removed constraint. In the similar way to adding a constraint, the algorithm reverses the dependency between the cell with such constraints and the cell that has been contained the removed constraint.

## 3 Implementation

Based on the algorithm presented in the previous section, we implemented the *DETAIL* constraint solver in Objective-C. It consists of two layers, a *solver* and *subsolvers*. A *solver* produces correct solution graphs, to which it applies local propagation. *Subsolvers* obtain values of variables by solving constraint systems locally in individual constraint cells. During local propagation, the *solver* invokes appropriate *subsolvers* based on solution types of constraints in cells. For example, if a cell contains only locally-predicate-better constraints, the *solver* calls the *subsolver* for $\tau_{\mathrm{LPB}}$. This architecture enables us to introduce a new solution type of constraints by implementing necessary *subsolvers*. For example, if we employ locally-predicate-better and least-squares-better constraints, we have to implement the subsolvers for $\tau_{\mathrm{LPB}}$, for $\tau_{\mathrm{LSB}}$, and for $\tau_{\mathrm{LPB}}$ and $\tau_{\mathrm{LSB}}$.[6]

We implemented three *subsolvers*: one that handles locally-predicate-better constraints represented as linear equations or multi-way constraints, one that treats least-squares-better linear-equation constraints, and one that generates graph layouts based on the spring model [1].

## 4 Performance Measurements

Using the chain benchmark [5], we compared the performance of the *DETAIL* constraint solver implemented in Objective-C with that of the DeltaBlue constraint solver implemented in C. Initially, the constraint hierarchy contains the required constraints $x_0 = x_1$, $x_1 = x_2, \ldots, x_{n-2} = x_{n-1}$ and the constraint weak stay($x_0$). The chain benchmark measures the planning time to add the constraint strong edit($x_{n-1}$) to the hierarchy, and also measures the execution time to compute values of variables when the value of $x_{n-1}$ is changed through edit($x_{n-1}$). Both of the planning and the execution are the worst cases where the overall solution graph must be processed.

Table 1 shows the result:[7] while the planning time of *DETAIL* is almost four times as long as that of DeltaBlue, the execution time is nearly twenty times as long. The main handicaps of *DETAIL* are the complex data structure of constraint cells, and dynamic binding of methods in Objective-C.[8] We believe that dynamic binding caused slowdown in performance because the source program involves numerous message sendings with dynamic binding. If we re-implement the *DETAIL* constraint solver in C++, its performance is expected to approach that of DeltaBlue.

[6] However, since the *DETAIL* algorithm tries to divide constraint hierarchies as much as possible, the subsolver for $\tau_{\mathrm{LPB}}$ and $\tau_{\mathrm{LSB}}$ may never be invoked.

[7] Precisely speaking, the separation of planning and execution is slightly different from the description presented in Sect. 2.4. In both *DETAIL* and DeltaBlue, the planning time includes the time of topological sort for local propagation.

[8] Objective-C does not support static binding like C++.

**Table 1.** Results of the chain benchmark

| n | | 1000 | 2000 | 3000 | 4000 | 5000 |
|---|---|---|---|---|---|---|
| Planning (ms) | *DETAIL* | 283 | 617 | 933 | 1183 | 1817 |
| | DeltaBlue | 67 | 166 | 250 | 350 | 434 |
| Execution (ms) | *DETAIL* | 36.7 | 68.3 | 105.0 | 140.0 | 176.7 |
| | DeltaBlue | 2.5 | 4.3 | 6.7 | 8.7 | 10.8 |

On NeXTstation *TurboColor* (33 MHz 68040)

## 5 Conclusions and Status

We proposed the *DETAIL* algorithm, which incrementally solves multiple solution types of constraints in a single constraint hierarchy by grouping together cyclic or conflicting constraints into constraint cells. We implemented the *DETAIL* constraint solver, which exhibited promising performance results.

Using this solver, we developed the IMAGE system, which generates GUIs by generalizing multiple visual examples [3]. This system takes advantage of the ability of *DETAIL* to handle hierarchies of simultaneous constraints. Also, we are planning on applying *DETAIL* to our algorithm animation system based on declarative specification [6].

## References


1. Kamada, T., *Visualizing Abstract Objects and Relations, A Constraint-Based Approach*. Singapore: World Scientific, 1989.
2. Maloney, J. H., A. Borning, and B. N. Freeman-Benson, "Constraint Technology for User-Interface Construction in ThingLab II," in *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1989, pp. 381–388.
3. Miyashita, K., S. Matsuoka, S. Takahashi, and A. Yonezawa, "Interactive Generation of Graphical User Interfaces by Multiple Visual Examples," in *Proc. of the ACM Symposium on User Interface Software and Technology*, Nov. 1994 (to appear).
4. Myers, B. A., D. A. Giuse, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *IEEE Computer*, vol. 23, no. 11, Nov. 1990, pp. 71–85.
5. Sannella, M., B. Freeman-Benson, J. Maloney, and A. Borning, "Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm," Technical Report 92-07-05, Department of Computer Science and Engineering, University of Washington, July 1992.
6. Takahashi, S., K. Miyashita, S. Matsuoka, and A. Yonezawa, "A Framework for Constructing Animations via Declarative Mapping Rules," in *Proc. of the IEEE Symposium on Visual Languages*, Oct. 1994 (to appear).
7. Wilson, M. and A. Borning, "Hierarchical Constraint Logic Programming," Technical Report 93-01-02a, Department of Computer Science and Engineering, University of Washington, May 1993.