A Generalized Algorithm for Solving Constraint Hierarchies[1]

Roman Barták

Department of Theoretical Computer Science
Charles University
Malostranské náměstí 25
Praha 1, Czech Republic

e-mail: `bartak@kti.mff.cuni.cz`
URL: `http://kti.ms.mff.cuni.cz/~bartak/`
phone: +42-2 21 91 42 42
fax: +42-2 53 27 42

Abstract

Constraint hierarchies have been proposed to overcome over-constrained systems of constraints by specifying constraints with hierarchical preferences. They are widely used in HCLP (Hierarchical Constraint Logic Programming), CIP (Constraint Imperative Programming) and graphical user interfaces. The advantages of constraint hierarchies are a declarative expression of preferred constraints and the existence of efficient satisfaction algorithms. At present, there exist a lot of relatively independent constraint hierarchy solvers/satisfaction algorithms that could be classified into two categories: refining and local propagation algorithms. While the local propagation algorithms are fast but limited to equality (functional) constraints the more general refining algorithms are not incremental.

In this paper we propose a generalized algorithm for solving constraint hierarchies. This algorithm combines advantages of both refining and local propagation approaches. It is based on ideas of local propagation however it is not limited to one type of comparators. The algorithm solves constraint hierarchies, even if some constraints must be solved simultaneously, by dividing them into constraint cells as much as possible. By constructing a constraint network/graph it also supports "constraint planning", i.e., the method of smart resatisfying of constraints when a value of one variable is changed. The proposed algorithm fits in our concept of plug-in architecture of constraint hierarchy solvers.

Keywords: constraint hierarchies, constraint solver, constraint network, plug-in architecture.

---

[1] The research was partially supported by the Grant Agency of Czech Republic under the contract No 201/96/0197.

1. INTRODUCTION

Constraint hierarchies were introduced for describing over-constrained systems of constraints by specifying constraints with hierarchical strengths or preferences[2]. Constraint hierarchies are widely used in areas like HCLP (Hierarchical Constraint Logic Programming) [8, 23] – an extension of CLP (Constraint Logic Programming) [13] to include constraint hierarchies, CIP (Constraint Imperative Programming) [14] – an integration of declarative constraint programming and imperative object-oriented programming, and graphical user interfaces construction [7]. The major advantage of constraint hierarchies is their declarative expression of preferences or strengths of constraints rather than encoding them in the procedural parts of the language.

In a constraint hierarchy, the stronger a constraint is, the more it influences the solution of the hierarchy. Additionally, constraint hierarchy allows "relaxing" of constraints with the same strength via weighted-sum, least-squares or similar methods.

Another important aspect of constraint hierarchies is also the existence of efficient satisfaction algorithms. Satisfaction algorithms, in other words constraint hierarchy solvers, can be classified into two groups: algorithms based on refining method and local propagation algorithms. Each group has its advantages and disadvantages but what they have in common is the ad-hoc method used for their construction. Almost all current constraint hierarchy solvers are designed for a specific comparator or for a certain type of constraints.

In this paper we propose a generalized algorithm for solving constraint hierarchies. This algorithm embraces both refining and local propagation concepts, hence it is at once enough efficient and satisfactory general. The algorithm solves constraint hierarchies, even if some constraints must be solved simultaneously, by dividing them into constraint cells as much as possible. By constructing a constraint network/graph it also supports "constraint planning", i.e., the method of smart resatisfying of constraints when a value of one variable is changed. The division of a constraint solving algorithm into two stages, i.e., the planing and the execution stages, is typical for local propagation and we preserve this feature in our algorithm too. Actually, we concentrate on the planing stage of the algorithm there as it is independent of the type of constraints and of the chosen comparator.

The proposed algorithm fits in our concept of modular plug-in architecture of constraint hierarchy solvers [1]. It means, e.g., that it is possible to construct various hierarchy solvers by adding modules with miscellaneous comparators and flat constraint solvers.

Although the proposed algorithm shares similar ideas with the DETAIL algorithm [10], it is completely different. While the DETAIL works with equality (functional) constraints and concentrates especially on removing cycles and conflicts from constraint graphs, we mainly focus on support of all types of constraints. Also, the DETAIL allows constraints with different strengths to be in one constraint cell, whereas our algorithm gathers only equally preferred constraints in the constraint cell. It admits the hypothesis that our constraint graphs are more structured, but then, we need more sophisticated execution phase.

[2] Another method for describing over-constrained systems is PCSP (Partial Constraint Satisfaction Problems).

The paper is organized as follows. In Sections 2 and 3, we present a short preview of the theory of constraint hierarchies followed by a brief catalogue of well-known constraint hierarchy solvers. The advantages and disadvantages of various constraint hierarchy solvers are also discussed there. In Section 4, we describe a general plug-in architecture of constraint hierarchy solvers. The Section 5 is dedicated to analysis of limits of local propagation. We introduce the notions of a constraint cell and of a constraint network in Section 6. We also give two sketches of planning algorithms for constructing constraint networks there. In Section 7, we give a preview of the execution algorithm that computes the evaluation of variables. We conclude with a summary of the paper.

2. CONSTRAINT HIERARCHIES

A theory of constraint hierarchies was developed in [6]. It allows the user to specify declaratively not only constraints that must hold, but also weaker, so called soft constraints at an arbitrary number of strengths. Weakening the strength of constraints helps to find a solution of previously over-constrained system of constraints. This constraint hierarchy scheme is parameterized by a comparator $\mathcal{C}$ that allows us to compare different possible solutions to a single hierarchy and to select the best ones.

Intuitively, the stronger a constraint is, the more it influences the solution of the hierarchy. Consider, e.g., an over-constrained system of two constraints: `x=0` and `x=1`. The user can attach a preference or strength to both constraints: `x=0@strong` and `x=1@weak`, and the arising constraint hierarchy yields the solution `{x/0}`. This property also enables programmers to specify preferential or default constraints those may be used in case the set of required, so called hard constraints is under-constrained (has more solutions). Moreover, constraint hierarchies allow "relaxing" of constraints with the same strength by applying, e.g., weighted-sum, least-squares or similar methods.

For purposes of the introduction to constraint hierarchies we will use the former definition of constraint hierarchies [8] which is simpler but also a bit different (e.g., it does not support regionally-better comparators) from the more recent definition [23].

A *constraint* is a relation over some domain $\mathcal{D}$. The domain $\mathcal{D}$ determines the constraint predicate symbols $\Pi_{\mathcal{D}}$ of the language. A constraint is thus an expression of the form $p(t_1,\dots t_n)$ where p is an n-ary symbol in $\Pi_{\mathcal{D}}$ and each t_i is a term. A *labeled constraint* is a constraint labeled with a strength, written $c@l$ where c is a constraint and l is a strength. The set of strengths is finite and totally ordered and it is usually given by the user.

A *constraint hierarchy* is a finite set of labeled constraints. Given a constraint hierarchy H, H_0 is a vector of required constraints in H, in some arbitrary order, with their labels removed. Similarly, H_1 is a vector of strongest non-required constraints in H up to the weakest level H_n, where n is a number of non-required levels in the hierarchy H. We also define $H_k=\emptyset$ for $k>n$. Note, that constraints in H_i are stronger (more preferred) than those in H_j for $i<j$.

A *valuation* for a set of constraints is a function that maps free variables in the constraints to elements in the domain $\mathcal{D}$ over which the constraints are defined. A *solution* to a constraint hierarchy is such a set of valuations for the free variables in the hierarchy that any valuation in the solution set satisfies at least the required constraints, i.e., the constraints in H_0, and, in addition, it satisfies the non-required constraints, i.e., the constraints in H_i for $i>0$, at least as well as any other valuation that also satisfies the required constraints. In other words, there is no valuation satisfying the required constraints that is "better" than any valuation in the solution set. Formally:

$$S_0 = \{ \theta \mid \forall c \in H_0 \;\; c\theta \text{ holds} \}$$
$$S = \{ \theta \mid \theta \in S_0 \;\&\; \forall \sigma \in S_0 \; \neg \; \text{better}(\sigma,\theta,H) \},$$

where S_0 is a set of valuations satisfying required constraints and S is a solution set.

There is a number of reasonable candidates for the predicate *better* which is called a *comparator*. We insist that *better* is irreflexive and transitive, however, in general, *better* will not provide a total ordering on the set of valuations. We also insist that *better respects the hierarchy*, i.e., if there is some valuation in S_0 that completely satisfies all the constraints through level k, then all valuations in S must satisfy all the constraints through level k:

$$\text{if } \exists \theta \in S_0 \; \exists k>0 \;\text{ such that } \forall i \in \{1,..,k\} \; \forall c \in H_i \;\; c\theta \text{ holds}$$
$$\text{then } \forall \sigma \in S \; \forall i \in \{1,..,k\} \; \forall c \in H_i \;\; c\sigma \text{ holds.}$$

To define various comparators we first need an *error function* $e(c,\theta)$ that returns a non-negative real number indicating how nearly a constraint c is satisfied for a valuation θ. The error function must have the following property:

$$e(c,\theta)=0 \Leftrightarrow c\theta \text{ holds.}$$

For any domain $\mathcal{D}$, we can use the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not.

Currently, there are two different groups[3] of comparators: locally-better and globally-better comparators. The *locally-better* comparators consider each constraint individually. They are defined by the following way:

$$\text{locally-better}(\theta,\sigma,H) \equiv_{\text{def}}$$
$$\exists k>0 \; \forall i \in \{1,\ldots,k\text{-}1\} \; \forall c \in H_i \;\; e(c,\theta)=e(c,\sigma) \;\&$$
$$\exists c' \in H_k \; e(c',\theta)<e(c',\sigma) \;\&\; \forall c \in H_k \; e(c,\theta) \le e(c,\sigma).$$

We can define a special type of locally-better comparator, *locally-predicate-better* ($\mathcal{LPB}$) comparator to be locally-better using the trivial error function.

The *globally-better* comparators combine errors of all the constraints at a given level H_i using a combining function g, and then compare the combined errors. They are defined as follows:

$$\text{globally-better}(\theta,\sigma,H,g) \equiv_{\text{def}}$$
$$\exists k>0 \; \forall i \in \{1,\ldots,k\text{-}1\} \;\; g(\theta,H_i)=g(\sigma,H_i) \;\;\&$$
$$g(\theta,H_k)<g(\sigma,H_k).$$

Using globally-better schema, we can define three global comparators, using different combining function g. This comparator triple enables the user to add a positive real number, called *weight*, to each constraint. Weights allow relaxing of constraints with the same strength then. The weight for constraint c is denoted by w_c.

---

[3] The recent definition of constraint hierarchies [23] also supports another type of comparator called regionally-better.

weighted-sum-better(θ,σ,H) $\equiv_{\text{def}}$ globally-better(θ,σ,H,g),

where $g(\tau, H_i) = \sum_{c \in H_i} w_c * e(c, \tau)$

worst-case-better(θ,σ,H) $\equiv_{\text{def}}$ globally-better(θ,σ,H,g),

where $g(\tau, H_i) = \max_{c \in H_i} \{ w_c * e(c, \tau) \}$

least-squares-better(θ,σ,H) $\equiv_{\text{def}}$ globally-better(θ,σ,H,g),

where $g(\tau, H_i) = \sum_{c \in H_i} w_c * e^2(c, \tau)$.

3. Constraint Hierarchy Solvers

An important aspect of constraint hierarchies is that there are efficient satisfaction algorithms proposed. We can categorize them into the following two approaches:

The *refining algorithms* first satisfy the strongest level, and then weaker levels successively.

The *local propagation algorithms* gradually solve constraint hierarchies by repeatedly selecting uniquely satisfiable constraints.

To illustrate both approaches consider, e.g., the following constraint hierarchy:

`x=y@required, x=z+1@strong, z=1@medium, x=1@weak.`

The refining algorithm first solves the required constraint `x=y` with the result `{x/V,y/V}`, followed by the strong constraint `x=z+1` leading to `{x/Z+1,y/Z+1,z/Z}`. Then, it evaluates the medium constraint `z=1` and gets solution `{x/2,y/2,z/1}`. Finally, it attempts to solve the weak constraint `x=1` but as it conflicts with the assignment generated by the stronger constraints, it remains unsatisfied.

By contrast, the local propagation algorithm first solves the medium constraint `z=1`, then propagates the value `{z/1}` through the strong constraint `x=z+1`, i.e., computes `{x/2,z/1}`, and, finally, through the required constraint `x=y`, i.e., `{y/2,x/2,z/1}`. Note, that the weak constraint `x=1` remains unsatisfied as it was rejected by the stronger constraints.

The refining method is a straightforward algorithm for solving constraint hierarchies as it follows the definition of solution, in particular the property of respecting the hierarchy. It means that the refining method can be used for solving all constraint hierarchies using arbitrary comparator. Its disadvantage is recomputing the solution from scratch everytime a constraint is added or retracted. The refining method was first used in a simple interpreter for HCLP programs [8] and it is also employed in the DeltaStar algorithm [23] and in a hierarchical constraint logic programming language CHAL. We show later (Section 6) that our generalized framework for solving constraint hierarchies covers the refining method.

Local propagation takes advantage of the potential locality of typical constraint networks, e.g., in graphical user interfaces. Basically, it is efficient because it uniquely solves a single constraint in each step (execution phase). In addition, when a variable is repeatedly updated, e.g., by user operation, it can easily evaluate only the necessary constraints to get a new solution. This straightforward execution phase is paid off by a foregoing planning phase that choose the order of constraints to satisfy.

Local propagation is also restricted in some ways. Most local propagation algorithms (DeltaBlue [18], SkyBlue [17], QuickPlan [20], DETAIL [10], Houria [9]) can solve only equality constraints, e.g., linear equations over reals. The exception is the Indigo [5] algorithm for solving inequalities that combines local propagation and refining method. We borrowed the main idea behind the Indigo algorithm, i.e., the propagation of the set of values, to our algorithm. The local propagation algorithms also usually use locally-predicate comparator or its variant respectively. Only Houria III and DETAIL can use globally comparators and Indigo uses metric comparator. Finally, local propagation cannot find multiple solutions for a given constraint hierarchy due to the uniqueness.

4. A PLUG-IN ARCHITECTURE OF HCLP SOLVER

Almost all current constraint hierarchy solvers have at least one common property. They are constructed ad-hoc for a limited class of constraints and using only one particular comparator or a small group of similar comparators respectively. The common reason for doing it is the (mis)belief that a more specialized solver is also a more efficient one. In [1] we suggest to use a more general architecture of constraint hierarchy solver without loss of efficiency.

The structure of the proposed plug-in architecture follows directly from the definition of a constraint hierarchy solution. It extends and generalizes the architecture of the simple HCLP interpreter [8] for support of various comparators and flat constraint solvers. The following figure shows the plug-in architecture of a constraint hierarchy system.

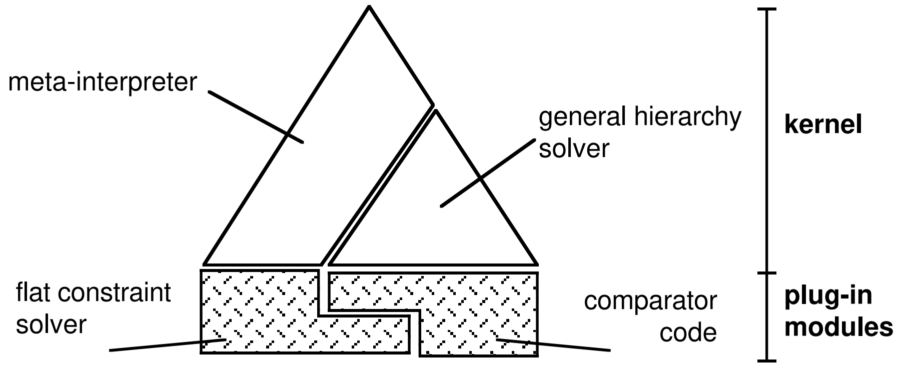

Figure 1 (plug-in architecture)

The *kernel* of the architecture includes a *meta-interpreter* or, more generally, a core system exploiting constraint hierarchies. The second part of the kernel is a *general hierarchy solver* which defines a universal method for solving constraint hierarchies. The general hierarchy solver should be independent of a chosen comparator and of a chosen set of constraints which depends on the domain $\mathcal{D}$. It reflects the method used for solving constraint hierarchies, i.e., currently the refining method or the local propagation respectively. We described a general hierarchy solver based on refining method in [1] and we concentrate on generalized algorithm for solving constraint hierarchies that is based on local propagation but also covers refining method in this paper

The *extension* part of the architecture forms a pair of plug-in modules. There is a plug-in module which implements a particular *comparator* and which closely cooperates with the second module, a *flat constraint solver*. The services provided by the flat constraint solver depend on demands of the comparator module.

The arrangement of elements in the Figure 1 is not self-involved. It expresses the binds and data-flow between individual components, i.e., the higher module calls services of the lower module(s).

In [1] we identified two typical services provided by the general hierarchy solver, i.e., adding a labeled constraint to a constraint hierarchy (planning) and solving a constraint hierarchy (executing). Although the parts implementing these two services do not call each other, they are closely related through sharing the data structure which describes the hierarchy of constraints. Hence, we usually do not separate them into two modules. However, according to the conception of data-flow between modules we can draw the structure of the general hierarchy solver in a following way.

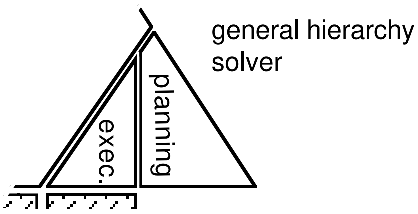

Figure 2 (structure of the general hierarchy solver)

Note, that the planning module does not communicate neither with the comparator module nor the flat constraint solver and thus it is fully independent of a chosen comparator and of a chosen set of constraints.

The names of the general hierarchy solver submodules correspond to the obvious phases of the local propagation solver. The planning and the execution phases can also be identified in our generalized algorithm.

The difference between the refining method and the local propagation in terms of planning and executing modules is the complexity of a particular module. While the refining method uses simple planning, i.e., distributing labeled constraints into levels according to strength [1], the local propagation algorithms employ a more complex planning phase which converts a set of labeled constraints into a constraint graph [18]. Contrary, the execution phase of the refining method is usually much more complicated that the execution phase of the local propagation. The generalized algorithm for solving constraint hierarchies, which is presented in following sections, is flexible enough to scale between these two extremes.

5. LOCAL PROPAGATION LIMITS IN DEPTH

When we investigated the limits of the local propagation algorithms we identify the following problems:

- solving conflicts among constraints is sometimes inappropriate
- local propagation cannot handle cycles of constraints
- local propagation works only with equality (functional) constraints
- local propagation supports only locally predicate better comparators
- local propagation cannot find multiple solutions.

As the execution phase of the local propagation requires every variable to be computed by just one constraint, the planning phase has to choose among conflicting constraints which bound the variable. However, solving this conflict is sometimes impossible, e.g., when constraints have the same strength (x=1@strong, x=2@strong), and sometimes it is too restrictive, i.e., a weaker constraint (y=1@weak) is disabled (assumed unsatisfied) to enable satisfying of a stronger constraint (y=1@strong) even if the weaker constraint is also satisfied.

The execution phase of the local propagation is a linear process. It means that when a constraint computes the value of one of its variables, the values of all other variables in the constraint have to be known, i.e., the values of these variables have had to be computed by other constraints before. This feature disables solving the set of constraints containing the same variables, e.g., the system of equations (x+y=3, x-y=1). Such a system of constraints corresponds to the cycle in the constraint graph, hence we speak about cycles of constraints. Some local propagation algorithms solve constraint cycles by evoking an external solver [7].

We mentioned the way a constraint is used to compute the value of one of its variables in the above paragraphs. The constraint is assumed there to be a function that computes the value of the output variable from the values of input variables. However, this approach disables many types of constraints like inequalities.

Every constraint, which is used in the execution phase, is completely satisfied while other constraints are entirely disabled during the planning phase. It implies the application of the predicate type of comparator in the classical local propagation. As every constraint is considered individually in the constraint graph it indicates the usage of the locally-better comparator. Local propagation also cannot find multiple solutions due to the uniqueness of satisfying constraints.

6. CONSTRAINT NETWORKS AND PLANNING

By addressing problems of the local propagation we made the first step to improve the generality of local propagation algorithms. To eliminate most of the mentioned problems we introduce *constraint cells* which can contain more than one constraint and which have different functions. The constraint cell containing more constraints can easily handle conflicts between constraints with the same strength as well as it can naturally manage the constraint cycles (Figure 3). By encapsulating the constraints into a constraint cell we also enable using of more types of comparators including globally-better ones.

We suggest several types of constraint cells. There are "normal" cells, called *functional cells*, containing only one functional constraint that can uniquely compute its output variable(s) from input variables. These cells are the only one enabled by the classical local propagation and the constraints in these cells are known to be completely satisfied independently of the values of the input variables. Then, there are "generalized functional" cells, called *generative cells*, containing constraints which can propage sets of values of input variables to the set of values of output variable(s). As they can generate a set of values they enable finding multiple solutions. Nevertheless, it is also possible that a constraint in the generative cell is not satisfied according to the values of the input variables. Finally, we introduce *test cells* which, rather that computing a value of any variable, test the satisfiability of constraint(s) according to given values of input variables.

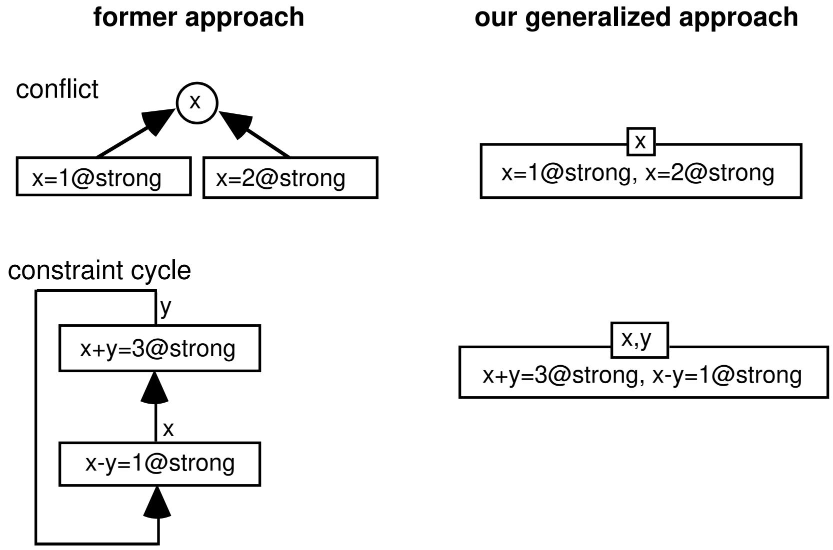

Figure 3 (removing conflicts and constraint cycles)

Individual constraint cells are connected into a constraint network similar to constraints graphs from classical local propagation. This network is created and maintained by the algorithm of the planning phase. Note, that this algorithm is completely independent of a particular type of constraints or used comparator. Due to a more complex structure of the constraint cell we shall also need a more sophisticated algorithm of the execution phase. This algorithm will use a particular comparator and constraint solver to find a solution by tracing the constraint network. Before we proceed to the formal definition of the constraint cell and related notions we depict some examples of constraint cells (we omit the strengths of constraints in the figure). Note that every constraint, even the equality, can be in a generative or a test cell.

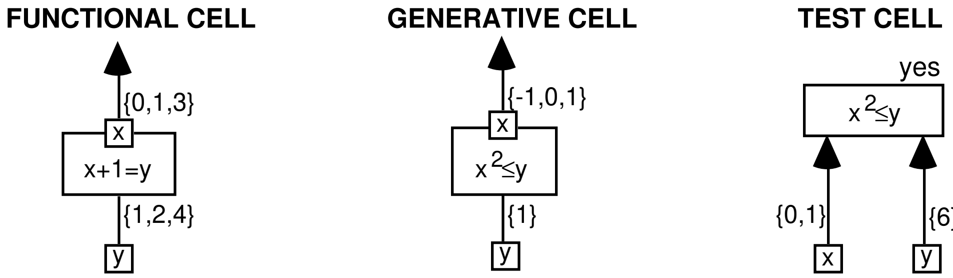

Figure 4 (types of constraint cells)

DEFINITION 1: (constraint cells)

Let C is a finite non-empty set of labeled constraints with the same strength and V is a set of all variables in constraints from C. For arbitrary sets of variables In,Out$\subseteq$V such that In$\cup$Out=V and In$\cap$Out=$\emptyset$ we define a *constraint cell* as a triple (C,In,Out). For every variable v we define a *constraint cell* ({},{},{v}) containing only the output variable v.

We call the sets In and Out from the constraint cell (C,In,Out) *input* and *output variables* respectively.

We also say that constraint cell (C,In,Out) *determinates* each variable from the set Out.

DEFINITION 2: (classification of constraint cells)

We classify constraint cells into the following groups:

- *free variable* ({},{},{v})
- *functional constraint cell* ({c@l},In,Out) such that for arbitrary evaluation θ of variables from In there exists a unique valuation σ of variable(s) in Out such that $c\theta\sigma$ holds
- *generative constraint cell* (C,In,Out) such that $C\neq\emptyset$ and $Out\neq\emptyset$ and (C,In,Out) is not functional
- *test* (C,In,$\emptyset$)
- *undecidable constraint cell* is a generative constraint cell or a test

Free variables and functional cells are well known from classical constraint graphs while generative cells and tests are contribution of this work. A constraint in a functional cell is always satisfied but we cannot decide whether constraints in generative cells and tests are satisfied during the planning phase. Thus, we call undecidable both the generative and test cells.

DEFINITION 3: (internal strength)

The *internal strength* of the constraint cell (C,In,Out) is the strength of any constraint in C. The *internal strength* of the constraint cell ({},{},{v}) is 'free' which is the strength that is weaker than any other strength of constraints.

DEFINITION 4: (constraint network)

Let H is a constraint hierarchy, i.e., a finite set of labeled constraints, and V is a set of all variables in constraints from H. We call a pair (CC,E) a *constraint network* if the following conditions hold:

1) (CC,E) is a directed acyclic graph with nodes CC and edges E
2) CC is a finite set of constraint cells containing only constraints from H, i.e.,
   $\forall Cell\in CC$ such that $Cell=(C,In,Out)$ $\quad C\subseteq H$
3) every constraint from H is located in just one constraint cell, i.e.,
   $\forall c\in H\ \exists!\ Cell\in CC$ such that $Cell=(C,In,Out)\ \&\ c\in C$
4) every variable from H is determined by just one constraint cell, i.e.,
   $\forall v\in V\ \exists!\ Cell\in CC$ such that $Cell=(C,In,Out)\ \&\ v\in Out$
5) for every constraint cell Cell there exist edges in E directed from constraint cells determining the input variables of Cell, i.e.,
   $\forall Cell,Cell'\in CC$
   $Cell=(C,In,Out)\ \&\ Cell'=(C',In',Out')\ \&\ In\cap Out'\neq\emptyset\ \Rightarrow (Cell',Cell)\in E$
6) for every undecidable constraint cell there does not exist an upstream constraint cell which has the same or weaker internal strength, i.e.,
   $\forall Cell\in CC$
   Cell is undecidable $\Rightarrow$ $\forall Cell'\in CC$ such that there exists a directed path from Cell' to Cell (Cell' is upstream to Cell), Cell' has a stronger internal strength than Cell

7) there does not exist "downstream forking" in an undecidable cell directed to other undecidable constraint cells, i.e.,
$\forall$Cell,Cell'$\in$CC
Cell and Cell' are undecidable & there does not exist directed path neither from Cell to Cell' nor from Cell' to Cell
$\Rightarrow$
$\forall$Cell''$\in$CC such that Cell'' is upstream both to Cell and Cell', Cell'' is not undecidable (i.e., it is a functional constraint cell)

The first five points of the constraint network definition are obvious conditions from traditional constraint graphs extended to cover the constraint cells. Thus, the proposed constraint network is a generalization of the former concept of constraint graphs [9,17,18,20]. The new conditions 6 and 7 of the Definition 4 are the contribution of this work. They help us to keep linearity and thus effectiveness of the execution algorithm. As the execution algorithm, computing values of variables, traverses downstream the constraint network, it has to be sure that using a constraint cell to compute its output variables does not disable any stronger constraint later, i.e., downstream the network. The condition 6 preserve this feature. The condition 7 keeps up the linearity of the execution algorithm.

The following figure shows two constraint networks corresponding to the same constraint hierarchy. It implies that there can exist more sound planning algorithms which construct the constraint networks. While the net on the right corresponds to the refining method (all constraints of the same strength are in one cell), the left net is more structured and thus it can more exploit local propagation methods (viz. Section 7).

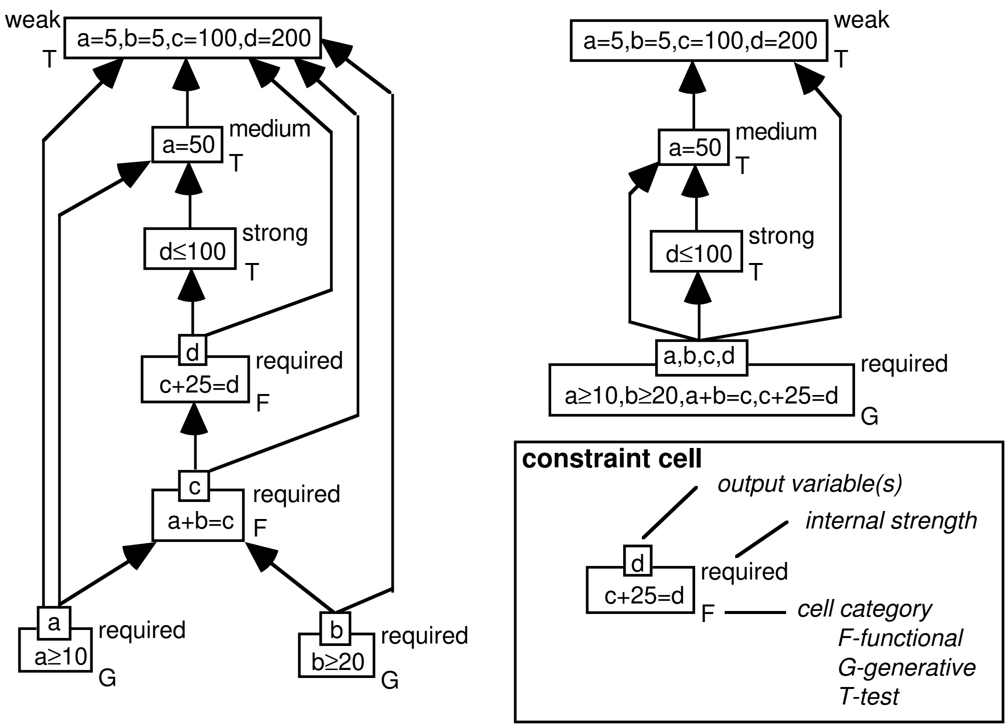

Figure 5 (constraint networks)

The constraint net is incrementally constructed by adding labeled constraints. This stage is usually called planning. We present two planning algorithms in the following paragraphs.

The first algorithm builds constraint nets similar to the right net in the Figure 5. This algorithm behaves in a following way. If there exists a cell with internal strength equal to the strength of the added constraint, then the algorithm adds the constraint into this cell. Otherwise, it creates a new cell containing this constraint. In the second phase the algorithm decides which variables of the added constraint are input and output respectively. Finally, it adds all necessary edges such that all conditions of the Definition 4 are satisfied. Note, that this algorithm does not use the free variable cells. While this planning algorithm is very simple, it requires the execution phase to mimic the refining method and, thus, to be ineffective.

Instead of formal description of the algorithm we give an example of adding a constraint to the net. The following figure shows the process of gradual addition of two constraints into the constraint net (read left to right).

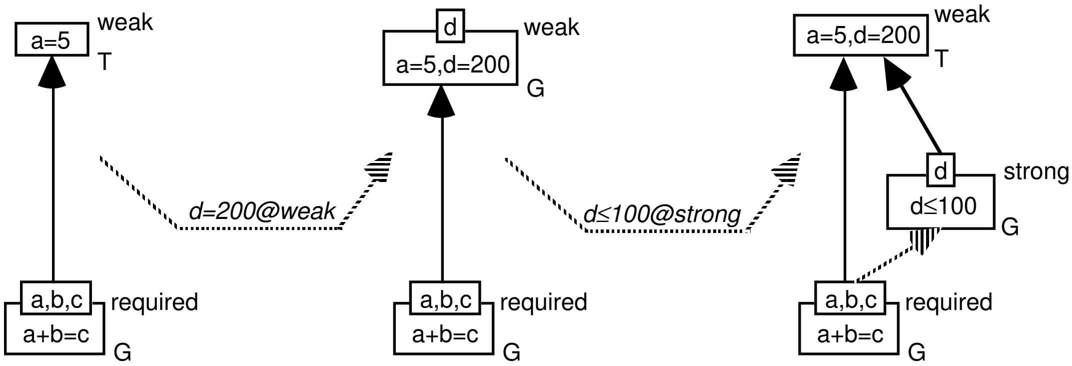

Figure 6 (constraint planning-refining method)

Note, that the shade edge between the required and the strong cell in the rightmost net is not entailed by the definition of the constraint network, nevertheless, this edge is also not explicitly forbidden by the definition. As we expect the execution algorithm to traverse the net from stronger to weaker cells (the refining method), the auxiliary edges can help to better navigate the constraint network.

The sophisticated planning algorithm, that builds structuralized constraint nets like the left net in the Figure 5, keeps the constraint cells as small as possible. This feature of the constraint network is desirable as it enables the execution algorithm to exploit the local propagation as much as possible. Lets us call this planning algorithm a gentle planner contrary to the raw planner that we described above.

The principle of the gentle planner is not complicated. First, the gentle planner tries to add a constraint as a new functional cell. If it does not succeed it adds a constraint as a new generative or test cell. Adding a constraint as a functional cell is almost identical to adding the constraint to a constraint graph using classical local propagation algorithm like DeltaBlue [18]. Nevertheless, we have to keep all conditions from the Definition 4 satisfied. In particular, we have to remove the downstream forking of undecidable cells that could possible arise after adding a new cell. The following figure sketches the process of removing the downstream forking.

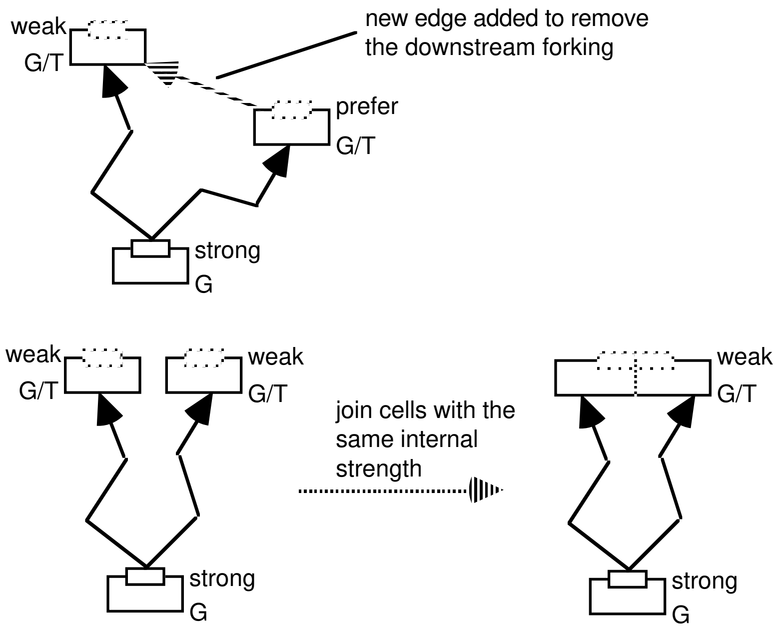

Figure 7 (removing downstream forking)

When it is not feasible to add a constraint as a functional cell, it is added as a generative or test cell. To satisfy the conditions 6 and 7 from the Definition 4 we possibly need to join some cells into one cell. The following figure shows example of adding constraints as generative cells.

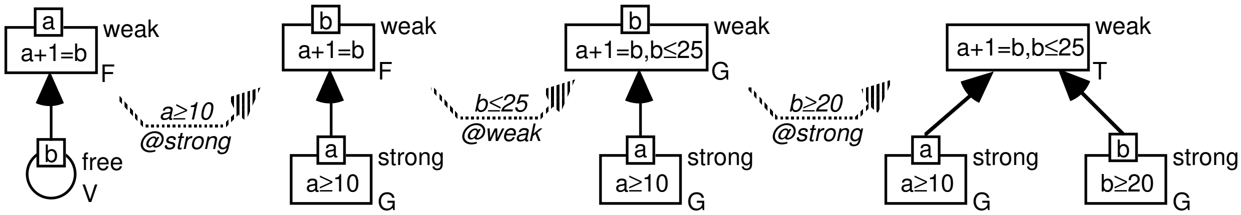

Figure 8 (constraint planning-gentle planner)

Some constraint cells can be deleted from the constraint net during the process of adding a constraint. Constraints from these cells are repeatedly added to the net till all constraints are in the net.

7. EXECUTION PHASE

In this section we will briefly demonstrate one possible algorithm of the execution phase of the generalized constraint hierarchy solver. We expect that many other algorithms of the execution phase can be developed to exploit the proposed structure of the constraint network.

The algorithm presented in this section is based on ideas behind the Indigo algorithm [5]. It means that the algorithm finds a solution of constraint hierarchy containing equality and inequality constraints over reals using locally-error-better comparator (i.e., locally better comparator that uses non-trivial error function). The proposed algorithm propagates set of values or intervals (in the weak version) through the constraint network in a similar way like the Indigo algorithm does. However, our algorithm is able to find multiple solutions contrary to the Indigo.

To demonstrate the algorithm we use the constraint hierarchy whose constraint network is depicted in the Figure 5 (the network on the left). First of all, the algorithm topologically sorts the constraint cells. In this particular case there are two orderings (left to right):

$a \geq 10$, $b \geq 10$, $a+b=c$, $c+25=d$, $d \leq 100$, $a=50$, $\{a=5, b=5, c=100, d=200\}$

$b \geq 10$, $a \geq 10$, $a+b=c$, $c+25=d$, $d \leq 100$, $a=50$, $\{a=5, b=5, c=100, d=200\}$.

We choose the first sequence and apply directly the Indigo algorithm to the first hexad of constraint cells from this sequence. As every cell in this hexad contains just one constraint we do not need to change the nature of the Indigo algorithm. We get the following partial solution [5]:

$a/50$, $b/\{20 \ldots 25\}$, $c/\{70 \ldots 75\}$, $d/\{95 \ldots 100\}$.

To get a final solution we choose an arbitrary ordering of constraints in the last constraint cell and apply the Indigo algorithm again. As it is possible to pick the ordering of constraints in the constraint cell it is conceivable to get multiple solutions. The following table shows all final solutions (valuations).

ordering of constraints in the last cell	solution
starts with $b=5$, e.g., $b=5$, $a=5$, $c=100$, $d=200$	$a/50$, $b/20$, $c/70$, $d/95$
starts with $c=100$ or $d=200$, e.g., $c=100$, $a=5$, $b=5$, $d=200$ or $d=200$, $a=5$, $b=5$, $c=100$	$a/50$, $b/25$, $c/75$, $d/100$

Note that applying the Indigo algorithm to the constraint cell containing more constraints is justified by using the locally better comparator. When we choose another type of comparator we have to use another method for solving such cells. However, the frame of the algorithm, i.e. propagation of sets of values through the constraint network, remains the same and thus the modularity of the algorithm is preserved (viz. Section 4).

In addition to finding multiple solutions, the sketched algorithm of the execution phase can solve the constraint hierarchy more effectively than the Indigo in some cases (the above example is not such a case). While the Indigo sorts constraints according to their strengths only, our algorithm uses topological ordering of constraint cells in the constraint network. Therefore, it is available to place as many as possible functional cells at the beginning of the sequence even if their internal strength is weaker than the strength of other cells and to place as many as possible of the rest functional constraints at the end of the sequence. Then, we can exploit the classical local propagation which is more effective than the Indigo, while the Indigo is used only in the central part of the sequence. The following figure shows such an ordering (the signs F, G and T denote functional, generative and test cells respectively).

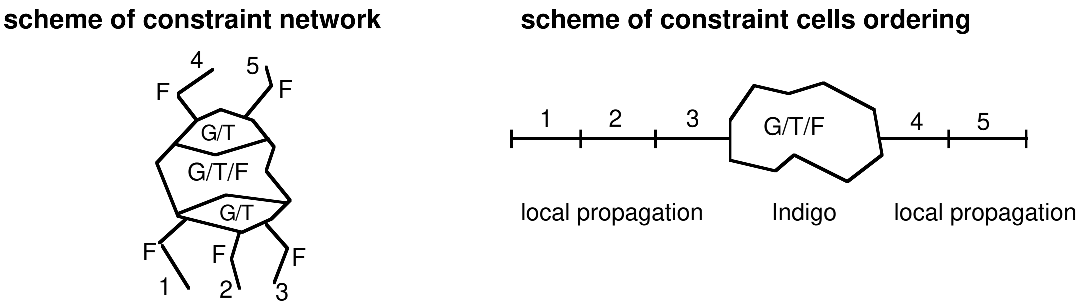

Figure 8 (smart ordering of constraint cells)

Note, that conditions 6 and 7 of the Definition 4 justify the usage of the Indigo algorithm in the central part of the sequence. These conditions ensure that all topological orderings of the constraint cells can be correctly used by the Indigo algorithm which requires stronger constraints to precede the weaker constraints.

CONCLUSIONS

In this paper we presented a generalized algorithm for solving constraint hierarchies and we showed how this algorithm fits in our concept of plug-in architecture of constraint hierarchy solvers. We addressed some drawbacks of classical local propagation algorithms and, consequentially, we defined more general notions of constraint cell and constraint network. The proposed concept of constraint network is a generalization of former constraint graphs. In particular, we concentrated on the planning phase of the proposed algorithm that constructs the constraint network. We also sketched an algorithm of the execution phase that is based on ideas behind the Indigo algorithm. The presented algorithms are instances of a general framework for solving constraint hierarchies based on constraint networks. This framework is a main subject of our future research.

ACKNOWLEDGMENTS

I would like to thank Petr Štěpánek for his continuous support, useful discussions and proof reading of the paper.

REFERENCES

[1] Barták, R., Plug-In Architecture of Constraint Hierarchy Solvers, Tech. Report No 96/8, Department of Computer Science, Charles University, December 1996

[2] Barták, R. and Štěpánek, P., Meta-Interpreters and Expert Systems, Tech. Report No 115, Department of Computer Science, Charles University, October 1995

[3] Barták, R. and Štěpánek, P., Mega-Interpreters and Expert Systems, presented at PAP'96, London, April 1996

[4] Benhamou, F. and Colmerauer, A. (eds.), *Constraint Logic Programming-Selected Research*, The MIT Press, Cambridge, Massachusetts, 1993

[5] Borning, A., Anderson, R., Freeman-Benson, B., The Indigo Algorithm, Tech. Report 96-05-01, Department of Computer Science and Engineering, University of Washington, July 1996

[6] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., Woolf, M., Constraint Hierarchies, in: *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp.48-60, ACM, October 1987

[7] Borning, A., Freeman-Benson, B., 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*, pp. 624-628, Cassis, France, September 1995

[8] Borning, A., Maher, M., Martindale, A., Wilson, M., Constraint Hierarchies and Logic Programming, in: *Proceedings of the Sixth International Conference on Logic Programming*, pp. 149-164, Lisbon, June, 1989

[9] Bouzoubaa, M., Neveu, B., Hasle, G., Houria III: Solver for Hierarchical System, Planning of Lexicographic Weight Sum Better Graph For Functional Constraints, in: *the Fifth INFORMS Computer Science Technical Section Conference on Computer Science and Operations Research*, Dallas, Texas, Jan. 8-10, 1996

[10] Hosobe, H., Miyashita, K., Takahashi, S., Matsuoka, S., Yonezawa, A., Locally Simultaneous Constraint Satisfaction, in: *Principles and Practice of Constraint Programming---PPCP'94 (A. Borning ed.), no. 874 in Lecture Notes in Computer Science*, pp. 51-62, Springer-Verlag, October 1994

[11] Hosobe, H., Matsuoka, S., Yonezawa, A., Generalized Local Propagation: A Framework for Solving Constraint Hierarchies, in: *Principles and Practice of Constraint Programming---CP'96 (E. Freuder ed.), Lecture Notes in Computer Science*, Springer-Verlag, August 1996

[12] Jaffar, J., Maher, M.J., Constraint Logic Programming: A Survey, in: Journal of Logic Programming 19, pp. 503-581, 1994

[13] Jaffar, J., Lassez, J.-L., Constraint Logic Programming, in: *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111-119, Munich, Germany, January 1987

[14] Lopez, G., Freeman-Benson, B., Borning, A., Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93 Virtual Machine, Tech. Report 94-07-07, University of Washington, July 1994

[15] Meier, M., Brisset, P., Open Architecture for CLP, TR ECRC-95-10, ECRC, 1995

[16] Menezes, F., Barahona, P., An Incremental Hierarchical Constraint Solver, in: [19]

[17] Sannella, M., The SkyBlue Constraint Solver, Tech. Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993

[18] Sannella, M., Freeman-Benson, B., Maloney, J., Borning,A., Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm, Tech. Report 92-07-05, Department of Computer Science and Engineering, University of Washington, July 1992

[19] Saraswat, V. and Van Hentenryck, P. (eds.), *Principles and Practice of Constraint Programming*, The MIT Press, Cambridge, Massachusetts, 1995

[20] Vander Zanden, Brad, An Incremental Algorithm for Satisfying Hierarchies of Multi-way Dataflow Constraints, Tech. Report, Department of Computer Science, University of Tennessee, March 1995

[21] Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, 1989

[22] Wilson, M., Borning, A., Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison, Tech. Report 89-05-04, Department of Computer Science and Engineering, University of Washington, July 1989

[23] Wilson, M., Borning, A., Hierarchical Constraint Logic Programming, TR 93-01-02a, Department of Computer Science and Engineering, University of Washington, May 1993