Plug-In Architecture of Constraint Hierarchy Solvers[1]

Roman Barták

Department of Theoretical Computer Science
Charles University
Malostranské náměstí 25
Praha 1, Czech Republic
e-mail: `bartak@kti.mff.cuni.cz`
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) – an extension of CLP (Constraint Logic Programming) to include constraint hierarchies, CIP (Constraint Imperative Programming) – an integration of declarative constraint programming and imperative object-oriented 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 the more general refining algorithms are not incremental.

In this paper we propose a general plug-in architecture of constraint hierarchy solvers. It is a modular architecture based on the notion of mega-interpreter. The proposed structure divides a system embracing constraint hierarchies into four independent parts: a meta-interpreter, a general hierarchy solver, a comparator part and, finally, a flat constraint solver. These parts communicate through exactly defined interface or, more precisely, each part provides a set of services to other components. Although the paper describes the plug-in architecture for the HCLP solver using refining method and predicate comparators, we expect it to be applicable to other constraint hierarchy solvers too.

Keywords: constraint hierarchies, constraint solver, mega-interpreter, 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 hierarchy is a method 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) [4, 21], CIP (Constraint Imperative Programming) [8] and graphical user interfaces construction [5]. 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 general plug-in architecture of constraint hierarchy solvers. The proposed architecture has a modular structure that allows programmers to design various constraint hierarchy solvers from standardized modules. It means, e.g., that an advanced but flat constraint solver can be easily extended to a constraint hierarchy solver by pushing it into the proposed structure. It is also easy to get constraint hierarchy solvers with miscellaneous comparators by exchanging the comparator module only. However, it should be noted that it is not possible to combine arbitrary modules, i.e., some modules (e.g., modules for metric comparators) need extended support from other modules (e.g., flat constraint solver).

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. In Section 4, we describe a proposed general plug-in architecture of constraint hierarchy solvers. There are examples of a PROLOG source code of both kernel modules (meta-interpreter, general hierarchy solver) and extension modules (flat constraint solver, comparator code) in two subsections of the Section 4. The presented code makes an HCLP solver over the Herbrand Universe using locally-predicate-better comparator. The paper is concluded with some remarks on the proposed architecture and with an appendix containing a PROLOG source code of another plug-in module that implements a weighted-sum-predicate-better comparator.

2. CONSTRAINT HIERARCHIES

A theory of constraint hierarchies was developed in [3]. 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

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

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 [4] which is simpler but also a bit different from the newer definition [21].

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, which must include "=". 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 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:

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

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:

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

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.

$$\text{weighted-sum-better}(\theta,\sigma,H) \equiv_{def}\ \text{globally-better}(\theta,\sigma,H,g),$$
$$\text{where}\ g(\tau,H_i)=\sum_{c\in H_i} w_c * e(c,\tau)$$

$$\text{worst-case-better}(\theta,\sigma,H) \equiv_{def}\ \text{globally-better}(\theta,\sigma,H,g),$$
$$\text{where}\ g(\tau,H_i)=\max_{c\in H_i}\{w_c * e(c,\tau)\}$$

$$\text{least-squares-better}(\theta,\sigma,H) \equiv_{def}\ \text{globally-better}(\theta,\sigma,H,g),$$
$$\text{where}\ g(\tau,H_i)=\sum_{c\in H_i} w_c * e^2(c,\tau).$$

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.

---

[3] The newer definition of constraint hierarchies [21] also supports another type of comparator called regionally-better.

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 [4] and it is also employed in the DeltaStar algorithm [21] and in a hierarchical constraint logic programming language CHAL. We choose the refining method for our general hierarchy solver (Section 4.1) to demonstrate practicability and generality of the proposed plug-in architecture.

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. 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. However, local propagation is restricted in some ways. Most local propagation algorithms (DeltaBlue [15], SkyBlue [16], QuickPlan [18], DETAIL [9], Houria [7]) can solve only equality constraints, e.g., linear equations over reals. The exception is the Indigo [6] algorithm for solving inequalities that combines local propagation and refining method. 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

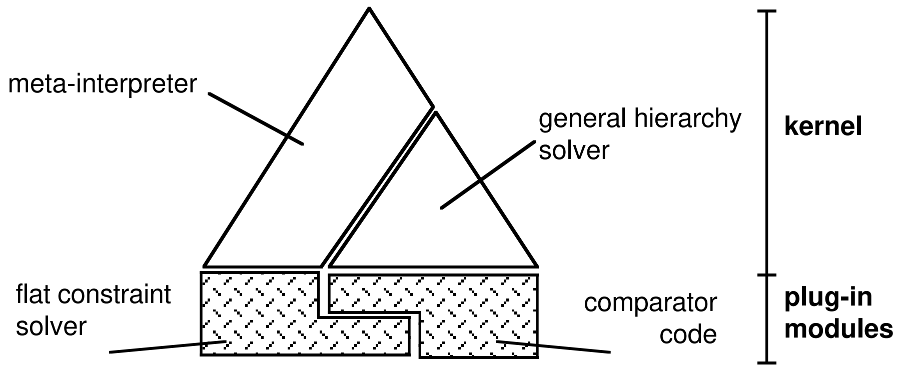
Almost all current constraint hierarchy solvers have at least one common property. They are constructed ad-hoc for a limited class of constraints and using only one particular comparator or a small group of similar comparators respectively. The common reason for doing it is the (mis)belief that a more specialized solver is also a more efficient one. Although we do not concentrate on efficiency issues of constraint hierarchy solvers in this paper, we suppose it is possible to use a more general architecture of constraint hierarchy solver without loss of efficiency. Our ideas are embodied in the plug-in architecture of constraint hierarchy solvers described hereinafter.

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 [4] for support of various comparators and flat constraint solvers. To express the ideas of proposed plug-in architecture we utilize the concept of a mega-interpreter [1, 2].

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.

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. Consider, e.g., an arbitrary predicate comparator. Then, a flat constraint solver should be able to solve a constraint according to the previously found solution or inform that the added constraint is not compatible with the current solution respectively (Section 4.2).

The following picture shows the proposed plug-in architecture of a constraint hierarchy system.



The arrangement of elements in the above picture is not self-involved. It expresses the binds and data-flow between individual components. Thus, the meta-interpreter asks for services the general hierarchy solver. The typical services provided by the general hierarchy solver are adding a labeled constraint to a constraint hierarchy and solving a constraint hierarchy. However, the meta-interpreter can also directly communicate with the flat constraint solver. This hand-to-hand cooperation is useful in case of solving required constraints and it speeds up the whole system so. We can also say that the meta-interpreter and the flat constraint solver form a non-hierarchical constraint system. If we omit the direct communication between the meta-interpreter and the flat constraint solver, we get a well-known scaled structure with the meta-interpreter at the top and the flat constraint solver at the bottom.

The general hierarchy solver uses services provided by the comparator component to pursue its tasks. The nature of these services will vary depending on used method. There will be a different set of required services for the refining method (Section 4.1) and the local propagation respectively. Note also, that the general hierarchy solver communicates with the flat constraint solver only through the comparator. Hence, the programmer of the general hierarchy solver does not use any information about the constraints and their domains, and by this way, he keeps the universality of the general hierarchy solver.

The data-flow between the comparator module and the flat constraint solver was discussed above.

Although, we will present a PROLOG code for an HCLP interpreter in the following sections, we realize that our plug-in architecture is usable for other constraint hierarchy systems as well. Only the core system embracing constraint hierarchies displace the meta-interpreter. The same applies to used method for a general hierarchy solver. We present the general hierarchy solver based on refining method there, however, the hierarchy solver using generalized local propagation [10] is almost finished.

4.1 THE KERNEL

To illustrate applicability of the plug-in architecture of constraint hierarchy solvers we implemented an HCLP interpreter using the proposed architecture. The kernel of the HCLP interpreter consists of:

- a meta-interpreter, much like traditional PROLOG meta-interpreters, and
- a general hierarchy solver based on refining method.

The *meta-interpreter* accepts a goal in form of the list of atomic goals and constraints and either satisfies it immediately, or looks up the goal in the rule base, reduces it to subgoals (`reduce`), and recursively solves the subgoals. Required constraints, possibly arising during unification[4], are passed on to the flat constraint solver immediately (`solve_constr`), while non-required constraints are passed on to the general hierarchy solver which simply push them onto a stack (`constr_to_hier`). When the goal is reduced, the collected constraint hierarchy is passed on to the general hierarchy solver (`solve_constr_hier`) along with the solution of required constraints, that has been found.

Labeled constraints are expressed in the form `C@L`, where `C` is a constraint and `L` is its strength. The set of strengths/labels is defined by the user using the fact `lab(OrderedListOfLabels)`, e.g., `lab([required,strong,medium,weak])`.

```
solve([C@L|T],CurrAnsw,Hier,Answ):-!,
      add_constr(C,L,CurrAnsw,Hier,NewAnsw,NewHier),
      solve(T,NewAnsw,NewHier,Answ).
solve([G|T],CurrAnsw,Hier,Answ):-
      reduce(G,CurrAnsw,NewG,NewAnsw),
      add_subgoal(NewG,T,NewT),
      solve(NewT,NewAnsw,Hier,Answ).
solve([],CurrAnsw,ConstrHier,Answ):-
      solve_constr_hier(ConstrHier,[CurrAnsw],Answ).

reduce(SG,OldAnsw,true,NewAnsw):-
      system(SG),!,
      copy_term(SG,CopySG),
      call(CopySG),
      solve_constr(SG=CopySG,OldAnsw,NewAnsw).
reduce(G,OldAnsw,NewG,NewAnsw):-
      copy_term(G,CopyG),
      clause(CopyG,NewG),
      solve_constr(G=CopyG,OldAnsw,NewAnsw).
```

[4] To simplify the code for presentation, we displace the unification of atoms $p(t_1,\ldots t_n)$ and $p(u_1,\ldots,u_n)$ by equality $p(t_1,\ldots t_n)=p(u_1,\ldots,u_n)$ instead of the list of equalities $t_1=u_1,\ldots,t_n=u_n$ in `reduce` procedure.

```
add_constr(Const,Lab,OldAnsw,OldHier,NewAnsw,NewHier):-
      lab(Labs),
      (Labs=[Lab|_]
            -> solve_constr(Const,OldAnsw,NewAnsw),
               NewHier=OldHier
               % required constraints are solved immediately
             ; constr_to_hier(Const,Lab,OldHier,NewHier),
               NewAnsw=OldAnsw).
               % soft constraints are collected
```

The next PROLOG code represents the implementation of the *general hierarchy solver* which provides two (relatively) independent services to the meta-interpreter:

- adding a non-required constraint to a constraint hierarchy (`constr_to_hier`) and
- solving a collected constraint hierarchy (`solve_constr_hier`).

The constraint hierarchy is kept as an ordered list of constraint levels during the goal reduction. The stacked constraint hierarchy is then solved using refining method, first satisfying the strongest level, and then weaker levels successively. Note, that there are two ways of finding a solution set:

- solutions are found one by one by the comparator module (`solve_level`) by means of backtracking mechanism ($\mathcal{LPB}$ comparator–Section 4.2)
- all solutions are found at once and then a particular solution/valuation is selected with the use of `select_answer` ($\mathcal{WSPB}$ comparator–Appendix)[5].

As the code is self-explaining, we hold off close comments.

```
constr_to_hier(Const,Lab,[C@L|T],[C@L|NewT]):-
      stronger(L,Lab),!,  % L is stronger than Lab
      constr_to_hier(Const,Lab,T,NewT).
constr_to_hier(Const,L,[Cs@L|T],[[Const|Cs]@L|T]):-!.
constr_to_hier(Const,Lab,T,[[Const]@Lab|T]).

solve_constr_hier([TopLevel@L|WeakerLs],PartAnsws,Answ):-
      solve_level(TopLevel,PartAnsws,SubAnsws),
      solve_constr_hier(WeakerLs,SubAnsws,Answ).
solve_constr_hier([],AnswList,Answ):-
      select_answer(Answ,AnswList).
```

The presented general hierarchy solver is really universal as it is absolutely independent of a chosen comparator. The next section is dedicated to implementation of a particular comparator and a flat constraint solver.

4.2 PLUG-IN MODULES

Thanks to the general structure of above defined kernel modules, it is possible to define various HCLP($\mathcal{D}$,$\mathcal{C}$) interpreters now. To get a complete HCLP($\mathcal{D}$,$\mathcal{C}$) interpreter that solves constraint hierarchies over the domain $\mathcal{D}$ using the comparator $\mathcal{C}$, we need to append to the kernel a pair of following extension modules:

- a flat constraint solver for solving constraints over the domain $\mathcal{D}$ and
- an implementation of selected comparator $\mathcal{C}$.

For purposes of this paper we choose the widely used *comparator*, locally-predicate-better ($\mathcal{LPB}$), which is easy and straightforward to implement. Example of

---

5 Also note that we do not include code for some auxiliary procedures like `add_subgoal` and `stronger`.

implementation of another comparator, weighted-sum-predicate-better ($\mathcal{WSPB}$), is given in the Appendix.

Note, that the solution of the constraint hierarchy found by the $\mathcal{LPB}$ comparator is the same as the solution of a maximal subset of satisfiable constraints on that level. The proposed algorithm (`solve_level_constr`) uses backtracking to find all maximal subsets of satisfiable constraints, i.e., to find all possible solutions of the constraint hierarchy level. As we use the predicate type of comparator (i.e., the trivial error function), the only service the comparator module asks for the flat constraint solver is solving a constraint according to the currently found solution (`solve_constr`).

```
solve_level(Cs,[PartA|_],[LevelAnsw]):-
      solve_level_constr(Cs,PartA,LevelAnsw).
solve_level(Cs,[_|PartAs],LevelAnsws):-
      solve_level(Cs,PartAs,LevelAnsws).

solve_level_constr([C|R],PartAnsw,Answ):-
      solve_constr(C,PartAnsw,SubAnsw),
      solve_level_constr(R,SubAnsw,Answ).
solve_level_constr([C|R],PartAnsw,Answ):-
      solve_level_constr(R,PartAnsw,Answ),
      not solve_constr(C,Answ,_).
            % solution can't be extended to cover C
solve_level_constr([],Answ,Answ).

select_answer(Answ,AnswList):-
      member(Answ,AnswList).
```

Note, that we have not decided yet the particular form of constraints and we have not specified the domain. So, the comparator module is (relatively) independent of a chosen domain and a chosen type of constraints. Therefore, it is possible to use other comparator module with the same flat constraint solver (see Appendix) as well as other flat constraint solver, e.g., over reals, with the same comparator.

Because of using the predicate type of comparators, the *flat constraint solver* provides just one service to other modules, i.e., solving a constraint according to the currently found solution. As we use PROLOG to implement the HCLP interpreter, we choose a natural domain for constraints in PROLOG, i.e., the Herbrand Universe. The Herbrand Universe enables two basic types of constraints, equalities and inequalities. We use an underlying PROLOG system to solve equalities, so, the valuation of free variables is implicit. As we also use inequalities, we add an explicit list of satisfiable but not satisfied, so called indeterminate, inequalities to solution. These inequalities are re-tested everytime a new variable is bound.

```
solve_constr(L=R,OldA,NewA):-
      L=R,    % equality test
      solve_constr_list(OldA,[],NewA).
            % re-test indeterminate inequalities
solve_constr(L\=R,OldA,NewA):-
      L\=R -> NewA=OldA         % inequality is satisfied
          ; (L==R ->  fail      % inequality fails
                ; NewA=[L\=R|OldA]).
                              % indeterminate inequality

solve_constr_list([H|T],OldA,NewA):-
      solve_constr(H,OldA,SubA),
      solve_constr_list(T,SubA,NewA).
solve_constr_list([],Answ,Answ).
```

Note again, that it is possible to use other flat constraint solver over arbitrary domain, e.g., reals, without changing the other modules of the architecture (with one exception, see footnote 4 above). The easiest way to do it is to use an underlying flat CLP($\mathcal{D}$) system.

CONCLUSIONS

In this paper we presented a general plug-in architecture of constraint hierarchy solvers. The main advantage of this approach is its modular structure that enables the programmer to develop various constraint hierarchy systems from standardized components. For this purpose we utilize the concept of mega-interpreter [1, 2].

The proposed architecture contains four modules which implement independent parts of the system embracing constraint hierarchies. The kernel of the architecture consists of the meta-interpreter and the general hierarchy solver. These components are independent of a particular comparator and of a particular domain for constraints which enables us to construct various constraint hierarchy systems by adding the following two plug-in modules: the comparator component and the flat constraint solver. The proposed architecture enables one to change various modules, e.g., the comparator component or the flat constraint solver, without affecting other parts of the architecture. Moreover, it enables to preserve the effectiveness of non-hierarchical constraint systems through the direct bind between the meta-interpreter and the flat constraint solver.

Although, we presented a plug-in architecture for an HCLP interpreter only, we realize that our architecture can be used for other constraint hierarchy systems as well. Similarly, the presented general hierarchy solver is based on the refining method and the presented flat constraint solver does not support other than predicate comparators. However, we are currently finishing implementation of a generalized local propagation [10] algorithm that is at least partially incremental, thus, more effective. We also expect that this generalized local propagation module enables us to develop an effective constraint hierarchy system with inter-hierarchy comparison [20].

The complete PROLOG source code of all above mentioned modules as well as the code for modules implementing other predicate comparators is available on-line at URL: `http://kti.ms.mff.cuni.cz/~bartak/prolog.html`.

REFERENCES

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

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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] Freeman-Benson, B.N., Borning A., Integrating Constraints with an Object-Oriented Language, in: *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pp. 268-286, 1992

[9] 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

[10] 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

[11] 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

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

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

[14] Menezes, F., Barahoma, P., Codognet, P., An Incremental Hierarchical Constraint Solver, in: *Proceedings of PPCP'93*, pp. 190-199, Newport, Rode Island, 1993

[15] 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

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

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

[18] 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

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

[20] 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

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

APPENDIX

To show the versatility of the proposed plug-in architecture we implemented various comparator modules including weighted-sum-predicate-better ($\mathcal{WSPB}$) comparator which is presented here. Note, that the weighted-sum-predicate-better comparator is a weighted-sum-better comparator using the trivial error function.

The module, that solves the given constraint hierarchy level using $\mathcal{WSPB}$, finds a maximal set of satisfiable constraints with the maximum sum of weights of these constraints. The labeled constraints are expressed in the form `(C,W)@L` where `C` is a constraint, `W` is its weight (a positive real number) and `L` is a strength[6].

Contrary to the $\mathcal{LPB}$ module, this module collects all solutions without using backtracking. Another difference from the $\mathcal{LPB}$ module is in collecting the solution in implicit form, i.e., the list of satisfiable equalities and inequalities, that is re-solved at the end (`select_answer`).

The $\mathcal{WSPB}$ module is based on the procedure `differ_answs` which differentiates the set of currently found partial valuations/answers into two sets, one set containing the valuations satisfying the added constraint C and the other set containing rest valuations.

```
solve_level(Cs,PartAnsws,NewAnsws):-
      pre_solve_level(Cs,PartAnsws,[],NewAnsws,0,0,_).

pre_solve_level(_,[],OldAnsws,OldAnsws,_,OldMax,OldMax):-!.
pre_solve_level([(C,W)|Cs],PartAs,OldAs,NewAs,
                                CurrMax,OldMax,NewMax):-
      differ_answs(C,PartAs,PartSatAs,_),
      NewWeight is CurrMax+W,
      pre_solve_level(Cs,PartSatAs,OldAs,PreAs,
                                NewWeight,OldMax,PreMax),
      pre_solve_level(Cs,PartAs,PreAs,
                                NewAs,CurrMax,PreMax,NewMax).
pre_solve_level([],CurrAs,OldAs,NewAs,CurrMax,OldMax,CurrMax):-
      CurrMax=OldMax,
      append(CurrAs,OldAs,NewAs),!.
pre_solve_level([],CurrAs,_,CurrAs,CurrMax,OldMax,CurrMax):-
      CurrMax>OldMax.
pre_solve_level([],_,OldAnsws,OldAnsws,CurrMax,OldMax,OldMax):-
      CurrMax<OldMax.

differ_answs(C,[A|As],RAs,[A|NAs]):-
      not solve_constr_list([C|A],[],_),!,
      differ_answs(C,As,RAs,NAs).
differ_answs(C,[A|As],[[C|A]|RAs],NAs):-
      differ_answs(C,As,RAs,NAs).
differ_answs(_,[],[],[]).

select_answer(A,As):-
      member(SelA,As),
      solve_constr_list(SelA,[],A).
```

The presented module was developed ad-hoc to test our ideas. Actually, we have not concerned ourselves yet with efficiency and this module is slow.

[6] Required constraints are expressed in obvious form `C@L` since the weight has no sense there.