Solution Strategies for Multi-domain Constraint Logic Programs

Stephan Frank, Petra Hofstedt, Peter Pepper, Dirk Reckmann

Berlin University of Technology, Germany
{sfrank,ph,pepper,reckmann}@cs.tu-berlin.de

Abstract. We integrate a logic programming language into Meta-S, a flexible and extendable constraint solver cooperation system, by treating resolution as constraint solver. This new approach easily yields a CLP language with support for solver cooperation.
Applying the strategy definition framework of Meta-S we define classical search strategies and more sophisticated ones, that allow an efficient evaluation of multi-domain constraint logic programs.

1 Introduction

To allow an efficient processing, in constraint programming the constraint solving algorithms are limited to restricted domains, e.g. linear or non-linear arithmetics, boolean constraints, finite domain constraints etc. However, many interesting problems are intuitively expressed as multi-domain descriptions. In multi-domain constraint problems every constraint may come from a different constraint domain and as well can be built itself using symbols of different domains. The cost and time for developing new constraint solvers that are able to handle such problems are significant. A different approach that has been researched actively during recent years is the collaboration of a number of individual (preexisting) constraint solvers managed by some meta mechanism which enables the cooperative solution of multi-domain constraint problems.

Meta-S [2] is a solver cooperation system embodying this approach. It allows the integration of arbitrary black-box constraint solvers and provides the user with a flexible strategy definition framework. In [3, 4] we theoretically discussed the idea of considering declarative programs (logic and functional logic ones) together with the associated language evaluation mechanism as constraint solvers and the integration of these solvers into our system. This yields multi-domain constraint programming languages and allows the definition of new evaluation strategies for them.

The present paper elaborates on this approach in practice. It is structured as follows: Section 2 briefly sketches our approach on solver cooperation. The integration of a logic language into our framework is explained in Sect.3. Using the strategy definition framework of Meta-S we define classical search strategies as well as new ones in Sect.4. Section 5 is dedicated to the evaluation and interpretation of experiments using these strategies. Finally, we draw a conclusion and discuss related work.

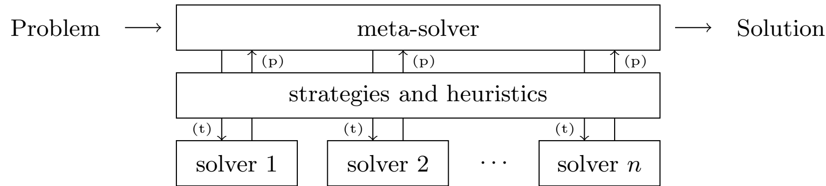

Fig. 1. Structure of Meta-S

2 Constraint Solver Cooperation

Our solver cooperation system Meta-S (for a detailed description see [2]) enables the cooperative solving of mixed-domain constraint problems using several specialized solvers, none of which would be able to handle the problem on its own. Such solvers can be e.g. a finite-domain solver, a boolean solver, or an interval arithmetic solver, but as well language evaluation mechanisms, like reduction or resolution as we will see in the following.

Fig.1 shows the structure of Meta-S. A coordination mechanism – the *meta-solver* – treats the different applied *solvers as black boxes.* It receives the problem to solve in form of mixed-domain constraints, analyzes them and splits them up into single-domain ones processable by the individual solvers. These problem constraints are held in a *global pool,* which is managed by the meta-solver. Constraints are taken from this pool and they are propagated to the individual solvers. These collect the constraints in their *constraint stores* ensuring at this their satisfiability. In return the solvers are requested to provide newly gained information (i.e. constraints) back to the meta-solver to be added to the pool and propagated to other solvers during the following computation. This communication is handled via the *solver interface* functions *tell* and *project*[1]:

- The function *tell* (denoted by (t) in the figure) for *constraint propagation* allows an individual solver to add constraints taken from the global pool to its store narrowing the solution set of the store's constraints.
- The *projection* function *project* (denoted by (p)) provides constraints implied by a solvers store for information interchange between the solvers. They are added to the pool.

The meta-solver repeats the overall *tell–projection* cycle until a failure occurs or the global pool is emptied. Problem solutions can then be retrieved by projection.

Meta-S provides a flexible *framework for strategy definitions* (including a strategy definition language) such that the user can formulate choice heuristics for constraints with respect to their structure and their domains, prescribe the order of propagation and projection, use constraint rewriting facilities and so forth.

We discuss the impact of the cooperation strategy in connection with the structure of the constraint problem at hand and the capabilities and efficiency of participating solvers on the effort of the solving process in Sect.5.

[1] These are based on typical solver functions, which for most preexisting solvers already exist or can be implemented by very little glue code, hence allowing a simple integration into our system. Their formal definitions are given e.g. in [4].

3 Considering a Logic Language as Constraint Solver

In [3] we presented a generalized method for the integration of arbitrary declarative languages and multi-domain constraints, we discussed this approach theoretically in detail in [4]. The idea is to consider declarative programs together with the associated language evaluation mechanism as constraint solver and to integrate this solver into our solver cooperation system.

It is widely accepted that logic programming can be interpreted as constraint programming over the Herbrand universe. Thus, by considering a *logic language as constraint solver* $CS_{\mathcal{L}}$, we can identify the goals according to a given logic program as the constraints handled by our new solver. Resolution as the language evaluation mechanism provides the constraint solving ability. This approach finally yields a CLP like language, where the constraints of other domains are collected by $CS_{\mathcal{L}}$ to be treated later by corresponding solvers within the Meta-S framework.

Let us consider the logic language solver $CS_{\mathcal{L}}$ in detail. For its integration into Meta-S we just need to define its *interface functions tell* and *project*.

Since it simplifies our presentation considerably, we write substitutions as special equations, e.g. $\sigma = \{x_1 = t_1, \ldots, x_m = t_m\}$ with variables x_i and terms t_i. Furthermore we make use of the usual notions, like the application $\sigma(e)$ of a substitution σ to a term e or the parallel composition $(\sigma \uparrow \phi) = mgu(\phi, \sigma)$ of idempotent substitutions, where mgu denotes the most general unifier (cf. [4]).

tell Resolution steps on goals correspond to constraint propagations. The thereby computed substitutions are collected in the constraint store $C_{\mathcal{L}}$ of $CS_{\mathcal{L}}$. Consider Fig.2. Our presentation of substitutions allows to consider the constraint store $C_{\mathcal{L}}$ of the solver $CS_{\mathcal{L}}$ directly as a substitution, which is written in the form of equations and thus can be treated like constraints.

Case 1a represents a successful resolution step. The result is a tuple of three values: First, the value *true* indicates success. The second value is a conjunction of constraints representing the constraint store after propagation (which remains unchanged). The third value is a disjunction of the instantiated bodies of the matching rules together with the computed substitutions σ_p which are returned back to the constraint pool for following propagations. If there is no applicable rule for a resolution step then the goal is rejected, indicating a contradiction by the first value $false$ (Case 1b).

Substitutions computed by resolution steps are given back to the pool (as equality constraints). A substitution $\phi = \{Y = t\}$ is propagated to $C_{\mathcal{L}}$ by parallel composition $(\phi \uparrow C_{\mathcal{L}})$ which may be successful (Case 2a) or failing (Case 2b).

This way the system performs a sequence of resolution steps and forwards the thereby computed substitutions via the pool to the store $C_{\mathcal{L}}$ to be collected.

project Projection is used for information interchange between the solvers which are integrated into the cooperation in Meta-S, here between $CS_{\mathcal{L}}$ and CS_{ν}, $\nu \in L$, where L denotes the solver indices.

The store of $CS_{\mathcal{L}}$ contains substitutions (received by resolution steps). Projection wrt. the domain of another solver CS_{ν}, provides an implication of the

tell: Let P be a constraint logic program, let $C_{\mathcal{L}} = \phi$ be the current store of $CS_{\mathcal{L}}$.

1. Let $R = p(t_1, \ldots, t_m)$ be the constraint (goal) which is to be propagated. Let $\hat{R} = \phi(R)$. We use the following notion: A rule $p = (Q_p \texttt{ :- } rhs_p)$ *applies to* $\hat{R}$, if there is a unifier $\sigma_p = mgu(\hat{R}, Q_p)$.
   (a) If the set $P_R \subseteq P$ of applicable rules is nonempty, then $tell(R, C_{\mathcal{L}}) = (true,\ C_{\mathcal{L}},\ \bigvee_{p \in P_R}(\sigma_p \wedge \sigma_p(rhs_p)))$.
   (b) If there is no applicable rule in P, then $tell(R, C_{\mathcal{L}}) = (false, C_{\mathcal{L}}, false)$.
2. Let $c = (Y = t)$ be the constraint (i.e. an equality constraint resp. substitution) which is to be propagated.
   (a) If $(\{Y = t\} \uparrow C_{\mathcal{L}}) \neq \emptyset$, then $tell(c, C_{\mathcal{L}}) = (true,\ \{Y = t\} \uparrow C_{\mathcal{L}}\ , true)$.
   (b) If $(\{Y = t\} \uparrow C_{\mathcal{L}}) = \emptyset$, then $tell(c, C_{\mathcal{L}}) = (false, C_{\mathcal{L}}, false)$.

---

project$_\nu$: The projection of a store $C_{\mathcal{L}} = \phi$ wrt. another solvers domain ν, $\nu \in L$, and a set of variables $X \subseteq X_{\mathcal{L}} \cap X_\nu$ makes the substitutions for $\texttt{x} \in X$ explicit:

$$project_\nu(X, \phi) = \begin{cases} \phi|_X & \text{if } \phi \neq \emptyset \\ true & \text{otherwise.} \end{cases}$$

Fig. 2. Interface functions *tell* and *project* of $CS_{\mathcal{L}}$

store $C_{\mathcal{L}}$, and thus simply generates equality constraints for the variables common to both solvers (cf. Fig.2).

Example 1. Consider the cooperation of our logic language solver $CS_{\mathcal{L}}$ within a framework of additional solvers, e.g. including an arithmetic solver $CS_{\mathcal{A}}$. A program describing electrical circuit problems may contain a rule for the sequential composition of resistors ($=_{\mathcal{A}}$ is the equality provided by $CS_{\mathcal{A}}$):
`r(seq(R1,R2),R) :- X + Y` $=_{\mathcal{A}}$ `R, r(R1,X), r(R2,Y).`

Let the pool hold the constraint $c =$ `r(seq(simple(A),simple(B)),600)` asking for a sequential composition of 600Ω of two simple resistors `A` and `B`. Let the store $C_{\mathcal{L}}$ of $CS_{\mathcal{L}}$ contain the equality constraint resp. substitution $\sigma = \{$`B = 200`$\}$ (computed during precedent resolution steps).

The *propagation of* c *wrt. the store* $C_{\mathcal{L}}$ by *tell* is done by performing a resolution step on $\sigma(c)$ (with most general unifier ϕ):
$tell(c, \sigma = C_{\mathcal{L}}) = (true, C_{\mathcal{L}}, \phi \wedge c')$ with
$\sigma(c) \rightsquigarrow_\phi c' =$ (`X + Y` $=_{\mathcal{A}}$ `600` $\wedge$ `r(simple(A),X)` $\wedge$ `r(simple(200),Y)`).

The constraint store $C_{\mathcal{L}}$ does not change. The constraint c is deleted from the pool and replaced by c' and ϕ derived from the resolution. Thus, in the next step an equality from ϕ or as well a constraint from c' could be chosen for propagation, e.g. `X + Y` $=_{\mathcal{A}}$ `600` for the arithmetic solver $CS_{\mathcal{A}}$ or `r(simple(200),Y)` (representing a logic goal) for a further resolution step. This choice depends on the cooperation strategy (cf. Sect.4).

Projection of the logic language store wrt. a set of variables means providing the collected substitution as equality constraints to other solvers. E.g. we project $C_{\mathcal{L}}$ wrt. `B` and the arithmetic solver: $project_{\mathcal{A}}(\{$`B`$\}, C_{\mathcal{L}}) =$ (`B` $=_{\mathcal{A}}$ `200`).

According to the described method, we plugged a new logic language solver, called CLL, into Meta-S. It quickly turned out that this method does not only allow the *integration of logic languages*[2] *with multi-domain constraint solving* but besides this the generation and experimentation with very different language evaluation strategies in an easy way. We consider this in the rest of the paper.

4 Solver Cooperation Strategies

We considered variations of the classical evaluation strategies *depth/breadth first search* for logic languages and two more advanced strategies, i.e. *lazy cloning* and *heuristic search.* Other search strategies like branch-and-bound, best-first, etc. can be easily integrated into our framework. However, we chose the above strategies to compare the typical PROLOG evaluation with more sophisticated strategies, which already have been proven to be advantageous in the context of constraint solver cooperation without language integration [2]. This comparison is instructive, because the differences between language evaluation and constraint solving influence the choice of an appropriate strategy for the problem at hand.

The solving process consists of a sequence of projection and propagation phases. In each of these phases, disjunctions may be returned by the constraint solvers, which evoke alternative branches in the search tree and, thus, make a cloning of the current system status (i.e. the pool and all stores) necessary. To cushion the exponential behaviour, such cloning steps should be avoided as much as possible. Therefore, our strategies perform *weak projections* which are only allowed to produce constraint conjunctions until a fixed point is reached. Then they switch to *strong projection* and allow the creation of disjunctions. E.g. for a store which allows only certain discrete values for a variable X, strong projection may express this by enumerating all alternatives, like $X = 1 \vee X = 3 \vee X = 7$, whereas weak projection results in a less precise conjunction $1 \leq X \wedge X \leq 7$.

Depth/Breadth First Search (dfs and bfs) These strategies are variants of the classical ones: A propagation phase performing depth/breadth first search until the pool is emptied alternates with a projection phase, where all solvers are projected and the results are added into the constraint pool.

For an illustration of the behaviour of these strategies see Fig.3a. To depict a configuration, we use tables showing the pool in the first row, and the solvers' stores in the second row, devided by vertical bars. Assuming a pool containing the conjunction $A \wedge B$ and a store containing the constraint S (further stores are not considered in this example), the next constraint propagated is A. Suppose that only a part of this constraint can be handled by the solver, i.e. A_3, and the disjunction $A_1 \vee A_2$ is put back into the pool. To ensure correctness, A must be equivalent to $(A_1 \vee A_2) \wedge A_3$. In the next step the configuration is cloned. To find all solutions of the constraint problem both resulting configurations must be processed further. This can be done by first processing the left one and after that the right on, which results in depth first search, or by interleaving steps with both configurations, which results in breadth first search.

[2] and, in general, arbitrary declarative languages

a) $\frac{A \wedge B}{S \mid \cdots} \rightsquigarrow \frac{(A_1 \vee A_2) \wedge B}{S \wedge A_3 \mid \cdots} \rightsquigarrow \frac{A_1 \wedge B}{S \wedge A_3 \mid \cdots} \vee \frac{A_2 \wedge B}{S \wedge A_3 \mid \cdots}$

b) $\frac{A \wedge B}{S \mid \cdots} \rightsquigarrow \frac{(A_1 \vee A_2) \wedge B}{S \wedge A_3 \mid \cdots} \rightsquigarrow \frac{(A_1 \vee A_2)}{S \wedge A_3 \wedge B \mid \cdots}$

c) $\frac{(A_1 \vee A_2 \vee A_3) \wedge (B_1 \vee B_2)}{S \mid \cdots} \rightsquigarrow \frac{B_1}{S \mid \cdots} \vee \frac{B_2}{S \mid \cdots}$

Fig. 3. Strategies: a) Depth/breadth first search b) Lazy cloning c) Heuristic search

Thereby, it is advantageous wrt. efficiency to abort the projection phase once a constraint disjunction is returned by any of the solvers, and then to reenter the propagation phase. This delays (often costly) projections when stores would be further restricted by propagating parts of the newly gained disjunction first.

Since depth first search and breadth first search traverse the same search tree, the measurements of computation steps in Sect.5 correlate; a small management overhead of breadth first search becomes obvious in the runtime measurements.

Lazy Cloning (lazy) The lazy cloning strategy described in [2] tries to increase performance by delaying cloning of the system state as much as possible. This approach is related to the Andorra principle as described in [10]. Whenever a disjunction appears, the current configuration is not cloned at once, instead the new disjunction is pushed into a queue and all other pending constraints are propagated first. When the queue of pending constraints is empty, a disjunction is dequeued and the configuration gets cloned.

The idea behind this strategy is to propagate common constraints only once. Otherwise, i.e. if configurations were cloned as soon as disjunctions appear, each clone would need to propagate the pending constraints.

Fig.3b shows an example using the lazy cloning strategy. The first step is the same as for depth first and breadth first search (cf. Fig.3a), but the next step differs. The cloning is delayed, and the constraint B is propagated first. In this example, the entire constraint B is assumed to be put into the store, and no disjunction is put back into the pool. Obviously, B is propagated only once, while it had to be propagated twice in Fig.3a.

Like the classical strategies, lazy cloning avoids projection as long as possible, which means propagating conjunctions first, then breaking disjunctions and switching to projection only when the constraint pool is completely empty. However, this avoidance of projection has a tradeoff: It implies a higher number of clones again, since a projection might disclose inconsistencies between two solvers *before* a queued disjunction is split into several clones.

Heuristic Search (heuristic) The heuristic search strategy is related to the fail first principle, proposed in [1], which tries to minimize the expanded search tree size and proved to be beneficial in solving constraint satisfaction problems. There, the idea is to choose the variable with the smallest domain size for value

assignment. This choice is the one most likely to fail and hence will cut away failing branches very fast. Furthermore, variables with large domains may get restricted further such that unnecessary search can be avoided.

The heuristic strategy uses this selection heuristic for choosing between several queued disjunctions while performing lazy cloning. We dequeue a disjunction with the least number of constraint conjunctions. As a further optimization we throw away all other disjunctions. To ensure that no knowledge is lost, all solvers are marked in order to project them later again. For illustration see Fig.3c. Here, the disjunction $(A_1 \vee A_2 \vee A_3)$ is removed, but can be reproduced later since it originates from a projection. Of course, this schema can be used only when propagation does not return disjunctions because they cannot be reproduced in this way. Thus, in general the heuristic strategy cannot be used in cooperations with the CLL solver, but in ordinary solver cooperations (cf.[2]).

Residuation Residuation is a method to cope with indeterminism caused by insufficiently bound variables, used in concurrent constraint languages. The concept of residuation, as defined for logic languages in [9], divides all rules into two groups. *Residuating rules* may be applied only when their application is deterministic, i.e. only one rule matches the goal that is subject to resolution. *Generating rules* do not need to be deterministic, but may only be used when no residuating rule is applicable. CLL supports both residuating and generating rules.

5 Evaluation

We present benchmark results of five illustrating examples (cf. Table 1) to investigate the effects of different strategies for multi-domain constraint logic problems with varying extent of logic predicates and multi-domain constraints.

The table captures the numbers of clones generated during the computation, of propagations and of projections. The propagations cover resolution steps and the propagation of equality constraints resulting from substitutions computed during resolution, both for CLL, as well as constraint propagation steps for other incorporated solvers. For a better discussion, the number of resolution steps is, however, pointed out in an extra column again. Projections are divided into weak and strong projections. Run time measurements are given merely as further additional information. Our solvers are proof-of-concept implementations and, their performance is, of course, not comparable to advanced solvers of modern systems like ECL^{i}PS^e Prolog. Nevertheless, transferring our strategies to such systems might improve their efficiency even further.

A Directed Acyclic Graph (`dag`) Our first example concentrates on logic language evaluation without constraints of other domains. It searches for paths in a directed acyclic graph with 13 nodes. Since a full graph traversal is necessary, all strategies create sooner or later all the clones for the possible edges. Thus, the number of clones (and for similar reasons, the number of projections) do not differ between the strategies.

The wanted effect of the lazy strategy first to propagate pending constraints of a conjunction before cloning stores (due to disjunctions produced by projec-

example	strategy	clones	prop.	resol.'s	w.proj.	s.proj.	time in s
`dag`	bfs/dfs	6143	24575	14335	24576	24576	21.74/19.02
	lazy	6143	22528	10241	24576	24576	18.05
`cgc`	bfs/dfs	9540	53426	12402	3027	3027	16.59/11.12
	lazy	9540	19082	12404	3027	3027	8.43
`hamming`	lazy	148	2141	895	266	266	0.85
`smm`	lazy	135	6550	19	456	148	1.38
	bfs/dfs	3	3333	19	902	105	1.34/1.32
`nl-csp`	lazy	35352	109852	0	285	192	55.33
	bfs/dfs	493	118751	0	34758	2614	49.38/47.82
	heuristic	345	109121	0	31485	4944	45.76

Table 1. Experimental results

tions), yields an explicitly smaller number of resolution steps (and thus propagations) here, and explains why lazy search is the fastest strategy in this scenario.

Path Costs in a Complete Graph (`cgc`) A complete graph (i.e. with bidirectional edges between every pair of nodes) is traversed. Again, the graph is described by logic facts, but this time each edge is annotated with random costs between 1 and 100. Rules for a path search additionally contain arithmetic cost constraints.

The structure and results of this problem are similar to the previous one. The only remarkable difference is the number of resolution steps, being nearly the same now for all strategies, although the number of overall propagations is lower for the lazy strategy. Again the reason is the delay of disjunction splitting (and thus the delay of cloning constraint stores) while preferring the propagation of pending constraints. While in the `dag` example this concerned goals, i.e. logic language constraints, in `cgc` arithmetic constraints are prioritized. This causes the significantly smaller number of of propagations for lazy search.

Experiments with different goal sequences in the rules of this example led to another observation (not documented in Table 1): Since the lazy strategy prefers conjunctions, while delaying the evaluation of disjunctive goals, it may reorder goals in the right hand side of rules in favour of the more deterministic ones. This causes situations, where the lazy strategy – "stepping over" a disjunction and evaluating deterministic goals first – terminates, while depth/breadth first produce infinite resolution derivations.

The Hamming Numbers (`hamming`) This example was chosen to show that Meta-S allows the combination of *residuating* and *nonresiduating* rules. A residuating rule computes (in cooperation with an arithmetic solver) multiples of 2, 3 and 5; its results are requested by nonresiduating CLL rules and merged into a common (here finite) list of hamming numbers.

In the table we only give the results for the lazy strategy. The other strategies behave similar because of the highly deterministic nature of the rule choice here.

Send-More-Money (`smm`) While it is well known that this problem can be solved by a usual finite domain solver providing arithmetics, we nevertheless found it appropriate as a short and simple example to illustrate general strategy effects for a solver cooperation of different solvers and domains.

The problem is constructed by a logic predicate describing the composition of letters to words and constructing the respective equality constraint, e.g. `S*1000+E*100+N*10+D=SEND`. This is done in a few deterministic evaluation steps, where only one rule definition at a time matches the goals to solve. Thus, the influence of language evaluation is surpassed by constraint solving. Besides the CLL solver two other solvers cooperate in this example. A linear arithmetic solver is supported by a finite domain solver. These two solvers rely on information exchange by projection, and hence the overall effort is greatly affected by the timing when switching from propagation to projection.

The tradeoff of projection avoidance for the lazy strategy which results in a higher cloning rate (as discussed in Sect.4) becomes obvious and hindering here. This shows, that mainly problems with an larger portion (and allowing alternative steps) of language evaluation than `smm` profit from lazy cloning.

A Nonlinear Constraint Satisfaction Problem (`nl-csp`) Another crypto-arithmetic puzzle: We want to replace different letters by different digits, s.t. `ABC * DEF = GHIJ` holds. This problem is given without any CLL rules.

The effect of an increased cloning rate for the lazy strategy gets enormous here. However, since the CLL solver is not used in this example, we may use the heuristic strategy, and the reordering of disjunctions is decisive in this case. This result corresponds to our observations in [2], where the heuristic strategy proved to be very efficient in many cases.

Overall Evaluation Our examples show that in problems with a large extent of logic language evaluation, the strategy lazy gives the best results. This strategy reorders goals in favour of deterministic ones and thus prevents a lot of propagations. An evaluation using the lazy strategy may even terminate in circumstances where depth/breadth first search produce infinite resolution derivations. In problems predominated by constraint solving most disjunctions are already avoided by the weak/strong projection schema. Hence the overhead of the lazy strategy outweighs its advantages here and depth/breath first search are more appropriate. Problems described without use of the CLL solver may benefit from the heuristic strategy, which gives near optimal results in many situations.

6 Conclusion and Related Work

We presented the integration of the logic language CLL into the solver cooperation framework Meta-S and the usage of its strategy definition framework to define very different evaluation strategies for the CLP language gained by this approach. We discussed implications and recommendations concerning the appropriateness of different cooperation strategies for the evaluation of multi-domain constraint logic programs of different form.

Todays solver cooperation systems often allow the definition of problem-specific cooperation strategies [5], or the integration of new solvers or both [7, 2]. Another concept for solver cooperation are computation spaces [8], which in contrast to more general systems rely on all solvers sharing the same store format.

This allows a tight solver communication but is not satisfying for the goal of Meta-S, i.e. cooperation of *black box solvers* independent of their implementation.

Meta-S comes with a strategy definition language which differs from other systems wrt. its language constructs (and finally offers the user all normal Common Lisp expressions). Besides this, it explicitly distinguishes between propagation and projection (in contrast to other approaches), which turned out to be useful and certainly important for defining efficient strategies. Meta-S already provides a set of predefined typical strategies.

The idea of integrating programming languages as constraint solvers into a solver cooperation system [3, 4] further distinguishes Meta-S from other approaches providing a fixed (often logic) host language. Using our cooperation approach, we achieve a covering of the well known scheme CLP(X) and as well CFLP(X) [6] for constraint functional logic programming.

References

1. J. Bitner and E. M. Reingold. Backtrack Programming Techniques. *Communications of the ACM (CACM)*, 18:651–655, 1975.
2. St. Frank, P. Hofstedt, and P.R. Mai. Meta-S: A Strategy-oriented Meta-Solver Framework. In *Proceedings of the 16th International Florida Artificial Intelligence Research Symposium Conference (FLAIRS)*. The AAAI Press, May 2003.
3. P. Hofstedt. A general Approach for Building Constraint Languages. In *Advances in Artificial Intelligence*, volume 2557 of *LNCS*, pages 431–442. Springer, 2002.
4. Petra Hofstedt and Peter Pepper. Integration of declarative and constraint programming. *Theory and Practice of Logic Programming (TPLP). Special Issue on Multiparadigm Languages and Constraint Programming*, 2006. to appear.
5. N. Kobayashi, M. Marin, and T. Ida. Collaborative Constraint Functional Logic Programming System in an Open Environment. *IEICE Transactions on Information and Systems*, E86-D(1):63–70, 2003.
6. F.-J. López-Fraguas. A General Scheme for Constraint Functional Logic Programming. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming – ALP'92*, volume 632 of *LNCS*. Springer, 1992.
7. B. Pajot and E. Monfroy. Separating search and strategy in solver cooperations. In *Perspectives of Systems Informatics – PSI*, volume 2890 of *LNCS*. Springer, 2003.
8. Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Computer Science*. Springer, 2002.
9. Gert Smolka. Residuation and Guarded Rules for Constraint Logic Programming. Technical Report RR-91-13, DFKI, 1991.
10. D.H.D. Warren. The Andorra Principle. Presented at the Gigalips Workshop, Swedish Institute of Computer Science (SICS), Stockholm, Sweden, 1988.