

System Description: Meta-S – Combining Solver Cooperation and Programming Languages

Stephan Frank, Petra Hofstedt, Dirk Reckmann

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

Abstract. Meta-S is a constraint solver cooperation system which allows the dynamic integration of arbitrary external (stand-alone) solvers and their combination with declarative languages. We sketch the main aspects of Meta-S including solver and language integration as well as its strategy definition framework for specifying solver cooperation and language evaluation strategies by means of an example.

1 Motivation

Constraint solvers offer problem solving algorithms in an encapsulated way. Many solvers have been designed and implemented, covering several different constraint domains, for example finite domain problems, linear arithmetic or interval arithmetic. However, many real world problems do not fit nicely into one of these categories, but contain constraints of various domains. Writing specialized solvers that can tackle particular multi-domain problems is a time consuming and error prone task. A more promising approach is the integration of existing constraint solvers into a more powerful overall system for solver cooperation. Meta-S – our flexible and extendable constraint solver cooperation system with support for integration of arbitrary declarative languages – implements this idea.

2 Meta-S

Constraint Solver Cooperation Figure 1 shows the general architecture of our constraint solver cooperation framework [3]. The system consists of several constraint *solvers*, each maintaining its own *store*. The collaboration is coordinated by the *meta solver* that establishes an exchange of information between the connected solvers. This meta solver maintains a *pool* of constraints to be solved.

The communication between the meta solver and the individual solvers is done solely through two interface functions (readily supported by many pre-existing solvers). A *propagation* function adds a constraint from the constraint pool into a solver's store, hereby ensuring consistency of the resulting constraint store. The second interface function handles *projection*, i.e. it infers knowledge implied by a constraint store in form of constraints that are put into the pool and passed to other solvers.

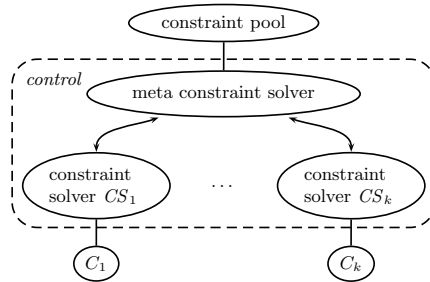


Fig. 1. Architecture of Meta-S

```

1  (define-meta-solver *smm*
2  (meta-solver eager-propagation-reordering)
3  ((my-fd fd-rational) ; solver integration
4  (my-lin cllin)
5  (my-cll cll-solver :file "smm.cll"))
6
7  (((in S,E,N,D,M,O,R,Y #{0 1 2 3 4 5 6 7 8 9}) ; constraints
8  (alldifferent {S E N D M O R Y})
9  (= (+ SEND MORE) MONEY)
10 (word ( [S,E,N,D] , SEND ))
11 (word ( [M,O,R,E] , MORE ))
12 (word ( [M,O,N,E,Y] , MONEY )))))

```

Fig. 2. The Send-More-Money problem specification

Example 1. To illustrate the usage of Meta-S, we consider the famous Send-More-Money problem.¹ Figure 2 shows the problem specification file. We use a collaboration of three individual constraint solvers: a *finite domain solver* for rational numbers (Line 3), a *linear arithmetic solver* (Line 4) and a constraint solver for *logic goals* (Line 5) based on a logic language (to be discussed later).

The problem to solve is given by a set of constraints (Lines 7-12). The domain constraint in Line 7 and the `alldifferent`-constraint in Line 8 are handled by the finite domain solver. The equation in Line 9 is handled by the linear arithmetic solver, and Lines 10-12 describe three goals for the logic language solver. The `word` predicate combines a sequence of digits to a number.

Language Integration In [2, 4] we show how to integrate declarative languages into Meta-S by treating the language evaluation mechanism as a constraint solver. This easily enables to integrate multi-domain constraints into a language.

Integrating a logic language into Meta-S yields *constraint logic programming*. Logic goals are evaluated by the constraint solver `cll` based on resolution.

¹ As is well known the problem can be solved by a usual finite domain solver with arithmetics. Nevertheless it is suitable here as a short and simple example.

```

1 (<- (word L X) (word L 0 X))           ; word(L,X) :- word(L,0,X).
2 (<- (word [] ACC ACC))                 ; word([], ACC, ACC).
3 (<- (word [FT|RT] ACC SUM))             ; word([FT|RT], ACC, SUM) :-
4   (word RT (+ (* 10 ACC) FT) SUM))     ; word(RT, 10*ACC+FT, SUM).

```

Fig. 3. Combination of letters to words (file `smm.cll`)

```

1 (== (word [] ACC) ACC)                 ; word [] ACC = ACC
2 (== (word [FT|RT] ACC)                 ; word [FT|RT] ACC =
3   (word RT (+ (* 10 ACC) FT)))         ; word RT 10*ACC+FT

```

Fig. 4. Functional logic combination of letters to words (file `smm.fc11`)

Consider again our example in Fig.2. Here the logic language solver was configured to read its rule definitions from the file `smm.cll` (cf. Fig.3). E.g. the goal `word([S,E,N,D],SEND)` is solved using the first rule for initializing an accumulator `ACC` with 0, calling thereafter the goal `word([S,E,N,D],0,SEND)`. The predicate `word/3` finally computes the value of the variable `SEND`. To ease the reading, the corresponding PROLOG rules are given as comments.

Of course, *Meta-S* allows the integration of other declarative languages as well. Consider for example the functional logic language solver *fell*. Its integration into *Meta-S* yields *constraint functional logic programming*. Our *fell* solver is based on narrowing using functional logic rules. Again the program must be given by the user in an extra file. E.g. Fig.4 shows the definition of a function `word/2` (equivalent to the predicate `word/3` in Fig.3). For better understanding the rules are given as comments in a HASKELL-like syntax. If we would have used the *fell* solver to describe the Send-More-Money problem, the corresponding constraints in the problem specification in Fig.2 would have been equality constraints like `(= (word [S,E,N,D]) SEND)`.

Problem descriptions may even freely mix logic predicates and functions, i.e. it is possible to integrate both the *c11* and the *fc11* solvers.

The Strategy Framework *Meta-S* provides the user with strategy definition mechanisms at two levels (cf. [1]). The definition of generic strategies on the first level mainly regulates the search tree traversal. On top of this, *Meta-S*' strategy specification language allows the user to define strategy attributes which concern the solver behaviour and their cooperation, like the order of propagation and projection, priorities of solvers within a cooperation or the usage of particular heuristics. Within the scope of the description of the solver behaviour, the user is able to specify and refine evaluation strategies for the language solvers, e.g. narrowing strategies. In [2] we compare variations of the classical evaluation strategies for logic languages (including residuation) as well as new advanced strategies in this context. *Meta-S* already comes with a collection of predefined search and cooperation strategies, including typical ones like depth-/breadth-first-search, eager and lazy narrowing and standard solver cooperation schemes.

```

1  (define-strategy eager-propagation-reordering (eager-strategy)
2    (:step (select ((eq-constr (= t t))
3                  (in-constr (in t t))
4                  (rest t))
5              (tell-all in-constr) (tell-all eq-constr) (tell-all rest)
6              (project-one linear-solver)
7              (tell-all) (project-all))))

```

Fig. 5. Reordering constraints for propagation

In the Send-More-Money example in Line 2 we state the usage of the strategy `eager-propagation-reordering`. Its definition in Fig. 5 illustrates the usage of the strategy definition language. This strategy is a refinement of the generic strategy `eager` (Line 1), which has a depth-first-search behaviour. The idea for the refinement is to classify the constraints according to their propagation cost (Lines 2-4) and to redefine the order of propagations and projections accordingly (Lines 5-7). This allows a distinct reduction of computation times (cf. [1]).

3 Conclusion

Besides solver cooperation systems with fixed solvers and fixed cooperation strategies, like [7], there are systems which allow a more flexible strategy handling, e.g. [5], up to the definition of problem specific solver cooperation strategies and the integration of new solvers, like the approach in [6] or `Meta-S`. Our system differs from others by its flexible coordination mechanism that is easily extensible by further solvers and makes it possible for users to adopt the solver cooperation strategy to the needs of the problem at hand. `Meta-S` furthermore allows the integration of declarative languages and multi-domain constraints and the definition and usage of new language evaluation strategies in an easy way.

References

1. St. Frank, P. Hofstedt, and P.R. Mai. `Meta-S`: A Strategy-oriented Meta-Solver Framework. In *Proc. of the 16th FLAIRS Conference*. The AAAI Press, 2003.
2. St. Frank, P. Hofstedt, and D. Reckmann. Strategies for the Efficient Solution of Hybrid Constraint Logic Programs. In *MultiCPL Workshop*, Saint-Malo, 2004.
3. P. Hofstedt. Cooperating Constraint Solvers. In *Sixth Conference on Principles and Practice of Constraint Programming - CP*, volume 1894 of *LNCS*. Springer, 2000.
4. P. Hofstedt. A general Approach for Building Constraint Languages. In *Advances in Artificial Intelligence*, volume 2557 of *LNCS*, pages 431–442. Springer, 2002.
5. E. Monfroy. *Solver Collaboration for Constraint Logic Programming*. PhD thesis, Centre de Recherche en Informatique de Nancy. INRIA, 1996.
6. B. Pajot and E. Monfroy. Separating search and strategy in solver cooperations. In *Perspectives of Systems Informatics - PSI*, volume 2890 of *LNCS*. Springer, 2003.
7. M. Rueher. An Architecture for Cooperating Constraint Solvers on Reals. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer, 1995.