Meta-S: A Strategy-Oriented Meta-Solver Framework

Stephan Frank and **Petra Hofstedt**
Technical University Berlin
{sfrank, ph}@cs.tu-berlin.de

Pierre R. Mai
PMSF IT Consulting
pmai@pmsf.de

Abstract

Meta-S is a practical implementation and extension of the theoretical framework developed by Hofstedt, which allows the user to attack problems requiring the cooperation of arbitrary domain-specific constraint solvers. Through its modular structure and its extensible strategy specification language it also serves as a test-bed for generic and problem-specific (meta-)solving strategies, which are employed to minimize the cooperation overhead incurred. This paper introduces Meta-S, focusing on its strategy-related aspects.

Introduction

Constraint solving has been actively researched during the last decades, and good progress has been made in reducing the computational costs associated with this approach for several specialized problem domains. However since many problems are more naturally described using mixed-domain constraints, one focus of current research has been on combining specialized constraint solvers into one solver that is able to handle such mixed-domain constraints. The work (Hofstedt 2001) describes a meta-solver approach that enables the cooperative solving of mixed-domain constraint problems using *arbitrary* specialized solvers, none of which would be able to handle the problem individually. Treating the employed solvers as black boxes, the meta-solver takes constraints from a global pool and propagates them to the individual solvers, which are in return requested to provide newly gained information (i.e. constraints) back to the meta-solver, through variable projections. In order to turn mixed-domain constraints into single-domain constraints, user and solver generated constraints are analyzed and split up into parts processable by the individual solvers. This overall *propagation–projection* cycle is repeated until a failure occurs or the pool is emptied.

The major advantage of this approach lies in the ability to integrate arbitrary, new or pre-existing constraint solvers, to form a system that is capable of solving complex mixed-domain constraint problems, at the price of increased cooperation overhead. This overhead can however be reduced through more intelligent and/or problem-specific cooperative solving strategies.

This paper describes the current implementation of the theoretical work of Hofstedt, which, taking aboard lessons learned in the first proof-of-concept implementation (Hofstedt, Seifert, & Godehardt 2001), offers a fully modular meta-solver framework, that also serves as a test-bed for ongoing research into new, more efficient generic solving strategies. It also provided the foundation for the design and implementation of an extensible strategy specification language, which allows users to define problem-specific solving strategies, based on their knowledge of the problem structure and/or the performance characteristics of the participating solvers. We will focus on the strategy-related aspects of Meta-S in this paper.

The following section gives a short overview of the framework, followed by a more detailed description of the strategy-related parts of the system, with special emphasis on those items that enable the integration of new solving strategies at various levels of the design. This leads over to the next section, where we are focusing on generic strategies, their construction and properties. The last section introduces the strategy specification language, including an example strategy. Finally we conclude our paper.

The Framework

The overall structure of the meta-solver framework, as depicted in Figure 1, was kept highly modular, to ensure that even radical changes to one module would not harm the utility of the overall framework. The three main modules are the *Base* module, the *Meta* module, and any number of pluggable constraint solvers.

The *Base* module provides the common substrate of the whole Meta-S system. This includes the external and internal representations of constraints, facilities for translating between those, a syntax extension facility, a pattern matching facility for constraints, and the abstract interface between the framework and attached constraint solvers.

As the system is embedded into the Common Lisp (CL) programming language, it was deemed most convenient to use an s-expression based syntax as the external representation for constraints. This has the additional benefit of eliminating the problems associated with operator precedence rules in a system with extensible operators, since s-expressions are fully parenthesized. In order to support the use of better-suited or more natural syntax for certain prob-

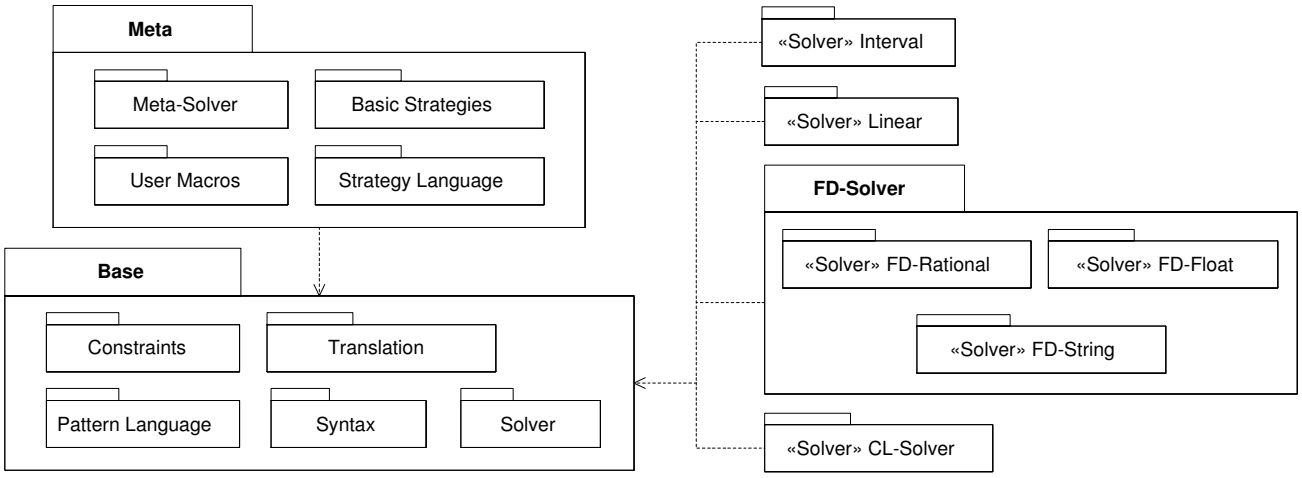

Figure 1: The module structure of Meta-S

lem domains, the system provides a syntax extension facility, which, based on CL reader-macros, allows the definition and selection of named syntax extensions.

The pattern matching facility provides positional pattern expressions for matching constraints based on their structure and contents. It is applied mainly in the strategy language, but also has uses in other parts of the system.

Using the abstract solver interface of the base module, we have currently interfaced several external and internal constraint solvers: linear arithmetic and interval arithmetic solvers, finite-domain solvers for rationals, floats and strings, and a CL solver toolkit that enables the generation of residuation-based solvers from CL functions.

The *Meta* module includes the meta-solver proper, user facilities to support the command line interface, and programmatic macros to ease the integration of the meta-solver system into other applications. Another part is the strategy framework, which will be considered later.

The class *meta-solver* lies at the core of this module. Instances of it represent each actual instantiation of the meta-solver system, which is defined by the participating solvers, the defined variables and the initial disjunction, i.e. it corresponds to one particular problem description and hence a complete constraint system that is to be tackled.

Hofstedt introduces the notion of *configuration* to represent the state of the architecture. The *meta-solver* class keeps all state information that is shared between all configurations. Instances of the class *meta-config* correspond to such configurations that each represent a potential solution. These configurations are necessary in order to process the different branches of a disjunction that each can individually fail or succeed. The state information of each configuration comprises the set of pending constraints (the constraint pool) and possibly some strategy specific information like the current projection mode[1], or pending disjunctions. The state information for the participating solvers like the store copy used in the current branch and the dirty flag (i.e. have there been changes since the last projection, which would necessitate further projections) is represented by instances of the class *solver-config*, one for each solver and configuration.

The meta-solver starts off with a configuration for each of the branches of the initial disjunction. From this set of pending configurations one is picked for the next solving step. Further disjunctions encountered during the solving process cause cloning of the current configuration, adding further configurations to the set of pending configurations. The configurations in this set are processed one by one until no pending configurations are left (all having either failed, thus causing their destruction, or terminated and hence moved to the set of *solved* configurations).

The abstract class *meta-config* keeps track of all associated *solver-config* instances, and offers a minimal interface between the configurations and the controlling *meta-solver*. This interface comprises the following categories of generic functions (GFs):

- GFs to control the life-time of configurations, i.e. creation of the initial configuration, cloning of configurations, and destruction once they have failed or are no longer needed.
- GFs to add new constraints and disjunctions (of conjunctions) of constraints to the pool of a configuration.
- A GF that controls the complete process of "solving" a particular configuration, which, once invoked by the meta-solver, continues processing until the configuration either fails (and is hence destroyed), succeeds, thus signifying a "solution" to the initial constraint problem, or is suspended, thus yielding control back to the meta-solver, without having completed processing.

[1] An optimization adopted from (Hofstedt, Seifert, & Godehardt 2001), which distinguishes between *weak* projection, which tries to avoid returning disjunctions where possible, e.g. by replacing them with interval constraints, and *strong* projection, which does not, in order to avoid backtracking induced by early creation of (large) disjunctions. Solvers are free to handle both like *strong* projection.

This last option can be used by strategies to e.g. limit processing depth/time per configuration, which can speed up the task of finding a single solution, quickly, rather than all possible solutions. A suspended configuration can be restarted at any time by reinvoking this GF.

- A GF that extracts the set of final constraints from a solved configuration for result presentation/processing.

As long as implementors adhere to the contracts underlying those generic functions, they are free to employ any particular solving strategy, while retaining the ability to use the remainder of the framework (i.e. the base constructs, the *meta-solver* and all attached solvers) without any changes.

Thus *meta-config* represents the natural place for implementing different solving strategies, through complete implementations of the GF *meta-config-run*. While this gives strategies much freedom in their implementations, it also means that they must implement nearly all code from scratch. Further experimentation showed that more code reuse could be achieved by factoring out more code that is common to many/most strategies, so that strategies could override default code at a finer granularity, thus keeping most of the freedom, but reducing the amount of work needed to implement new strategies. This led to the creation of the abstract class *strategy*, which we will describe next.

Generic Strategies

Central to the creation of the *strategy* class was the formulation of algorithm-independent termination conditions, which could be kept track of without particular assistance from the strategy proper, thus freeing the implementors of strategies from keeping track of termination conditions themselves.

In addition to the state kept by *meta-config*, the class keeps track of the set of pending constraints, the set of delayed constraints[2] (per solver), the current projection mode (strong or weak), and the remainder of any *parked* disjunction. *Disjunction parking* is an optimization which allows the gradual dissolution of a disjunction, so that, for a disjunction of *n* branches, instead of cloning the current state *n-1* times all at once, we clone it just once, leaving the remaining *n-1* branches of the disjunction as a *parked disjunction* in the clone. This reduces the set of live configurations considerably, especially for disjunctions with many branches.

Using those pieces of state information we are able to formulate the following set of termination criteria.

Termination of the solving process of a particular configuration ensues when:

1. there remain no pending constraints in the pool,
2. there is no parked disjunction,
3. none of the solvers are marked dirty, i.e. there are no changes in the solver stores that have not yet been requested by projection, and
4. we are currently in strong projection mode.

[2]Solvers can delay the processing of constraints they are offered to a later point in time, for example when the constraint references variables whose domain is not yet known to the solver.

Since these criteria can be independently tested at any point during the solving process, we can "outsource" the termination testing from the solving strategy itself. Furthermore, updating of all required state can be done automatically in the various individual actions, as we shall see below. Taken together, this allows the formulation of strategies without explicit checks for termination. Since the conditions are fairly narrow, every strategy that eventually empties the pool and projects all solvers w.r.t. all variables as well as eventually switches to strong projection, will terminate regardless of any other actions or the order of actions it performs.

The class *strategy* also extends the interface of *meta-config*, providing more fine-grained action-oriented generic functions, and thus more hooks where derived classes can add or substitute their own, specific code. It also provides many default implementations for those GFs, and the GFs of *meta-config*, in order to factor out code common to most/all strategies. The generic functions fall into one of the following categories:

- GFs to keep track of the delaying and undelaying of constraints.
- GFs to propagate constraints, and project individual or all solvers against a set of variables, defaulting to all outstanding variables of the given solver.
- A GF that handles the conversion of projected constraints from the solver they originated on to all other solvers, and another GF that automatically extends this conversion to the level of disjunctions of conjunctions of constraints.
- GFs that are invoked by the control loop to test for termination (*strategy-finish*), and to carry out all actions that are needed when the control loop is to be restarted (*strategy-restart-actions*, see below).
- The GF *strategy-run* that constitutes the whole control loop of the strategy, invoked by a default method on the GF *meta-config-run*, which establishes constructs to dynamically exit from the overall control loop at any time, returning a return value indicating the reason for termination, i.e. either success, failure or suspension.
- The GF *strategy-step* that carries out the "normal" processing of the control loop, and which is the central place for strategies to hook into to influence the ordering and form of propagation and projection actions.

The default method for *strategy-run* now implements the following overall algorithm:

```
restart:
        invoke strategy-restart-actions
        until pending constraints = ∅ ∧
              ∀solver:¬dirty(solver)
         do
              invoke strategy-step
        end until
        if strategy-finish returns then
              goto restart
        end if
```

Since individual strategies may employ optimizations that keep pending constraints from the set of pending constraints,

like e.g. the disjunction parking discussed earlier, we allow further termination checking to occur through methods on the generic function *strategy-finish*, which can restart the loop when needed. It does this by returning, and, in conjunction with methods on the GF *strategy-restart-actions*, by placing new constraints into the set of pending constraints, and/or causing the dirtying of solvers, so that the unaugmented termination check does not succeed.

Thus freed from keeping track of termination, the generic function *strategy-step* represents the place to implement strategies based on the order of constraint propagation and projection. The default method on this GF implements a simple propagate all, project all strategy.

Demonstrating the flexibility of the abstract *strategy* class we implemented three generic strategies that differ in their handling of constraint disjunctions. They were realized by defining methods on the generic functions for adding disjunctions and termination testing (*strategy-finish*), using less than 50 lines of code combined. Thus the strategies leave the overall control loop unchanged, and can be combined with problem-specific strategies defined using the strategy language described in the next section.

The individual strategies implement the following approaches to deal with disjunctions:

eager This is the most simple though surprisingly effective strategy. Once a disjunction is encountered a clone of the pool and the solver states is (eventually) produced for every disjunction branch. As a space optimization the actual clone creation for each branch is delayed until that branch is actually processed, keeping a backup copy for further cloning (see the discussion on *parked* disjunctions above).

All strategies share an optional variable projection order argument that influences the order in which variable projections are requested by the meta-solver, thus controlling the generation of disjunctions.

lazy This is similar to the eager strategy though cloning is delayed to process pending conjunctions in the hope of further pruning the search space with additional constraints before actually processing any pending disjunctions, thus performing the cloning this entails. However, as it turned out, this approach conflicts with the weak/strong-projection scheme, which already moves many disjunctions to a stage where projections do not yield any non-disjunctive constraints, thus removing most of the advantages of the lazy strategy.

heuristic This strategy is a hybrid of the previous two and integrates a heuristic element known as *fail first* as described in (Bitner & Reingold 1975). Since generic strategies in the meta-solver cannot have any domain knowledge as external solvers are treated as black boxes, they can only look at the domain size by counting the number of branches of a projected disjunction. Thus it is advantageous to perform strong-projections on all solvers, and then choosing the disjunction with the smallest set of possible values and discarding all other disjunctions. Care must be taken to mark solvers, which returned a now discarded disjunction, dirty (again) to ensure that the information they contain is requested again at a later stage.

strategy	time in s	# of clones	# of projections weak	strong
`lazy`	79.0	3521	18909	16126
`lazy`, ordered	27.7	5269	2761	25366
`eager`	91.2	2502	20933	2068
`eager`, ordered	9.2	253	2717	858
`heuristic`	10.2	73	3829	1276
`heuristic`, ordered	8.4	45	2915	1012
`domain-eq-first`	9.1	253	2728	858
`once-domain-eq-first`	9.2	253	2728	858
`eager-solver-flow`	6.3	253	2013	858
`heuristic-solver-flow`	5.4	45	2123	1012

Table 1: Benchmarking results for generic and problem-specific strategies for the well-known SEND+MORE=MONEY constraint problem.

This strategy uses the variable projection ordering as a tie breaker to choose among same-sized disjunctions.

These strategies were benchmarked against a set of more than 10 different constraint systems to evaluate their performance. The upper half of Table 1 gives their results for one particular constraint system, namely the well-known "SEND + MORE = MONEY" problem with multiple solutions, solved cooperatively by a linear arithmetic and a finite-domain solver. It illustrates the general performance levels of the generic strategies, where the "ordered" entries are based on strategies with a specified (near-)optimal variable ordering for the problem at hand, whereas the other entries use a random (identical) variable ordering.

The table underscores the important gains the heuristic strategy offers over the other strategies, because it usually (though not always) offers very good performance levels even in the absence of a known-good variable ordering, thus freeing the user from the need to find such a variable ordering for their specific constraint systems.

Strategy Language

Despite the quite comfortable performance improvements seen in the generic strategies, we have also seen that large gains can be achieved through the incorporation of problem-specific information (e.g. variable projection order) into the solving process. In order to allow the user to devise more efficient strategies based both on existing generic strategies, and his knowledge about the problem at hand, a strategy language was developed.

It was anticipated that the user would want to control the exact order of constraint propagation, based on the target solver and/or the form of the constraint, as well as the order of projection based on variables and solvers. Additionally, the user might, in certain circumstances, want to (re)write constraints on the fly, especially when converting projected constraints from one solver for the consumption of other solvers, which do not implement an identical set of relations or functions.

Founded on those considerations, the strategy language offers the following set of constructs:

- The language is designed as an extension to Common Lisp, thus offering all normal CL constructs to the user. This provides the user with all kinds of normal control and data flow constructs, as well as abstraction features like macros.
- Since strategies live at the core of the inner loop of constraint solving, efficiency of execution is very important. By compiling the strategy language (on invocation) via CL to native code, we ensure that no interpretation overhead is carried into the inner loop of execution.
- In order to allow the selection, destructuring and rewriting of constraints, the strategy language supports positional pattern expressions, with a corresponding pattern matcher. The pattern expressions allow destructuring of constraints by binding variables to matched subexpressions. There are also constructs to partition a set of constraints (e.g. the set of pending constraints) into disjoint subsets based on a set of pattern expressions.

  Like the strategy language as a whole, pattern expressions are compiled to efficient native code pattern matchers.

  Creation of constraints is supported through a template mechanism, based on the CL backquote reader-syntax.
- The language offers primitives for the propagation of sets of constraints, as well as the projection of individual or all solvers against a given set of variables.

Strategies are defined, based on existing strategies, using the *define-strategy* construct, which allows the user to specify methods to override those in the base strategy, through the specification of "method clauses". For example the *:step* "method clause" can be used to define methods on the generic function *strategy-step*. Another method clause is the *:convert* clause, which can be used multiple times in a strategy, to define more specialized methods for converting projected constraints between any pair of solvers in the system.

Using the constructs provided by the strategy language, it is possible to design more advanced, custom-tailored strategies, which easily outperform even the best generic strategies. Figure 2 gives the specification of one such strategy. This strategy prefers domain and equality constraints over other constraints, and takes advantage of the fact that it is mostly the linear arithmetic solver ("my-linar") which generates restrictions, whereas the finite-domain solver mostly just generates all possible combinations. The performance improvements thus gained can be seen in the lower half of Table 1, where it is compared to generic and other similar problem-specific strategies which prioritize different types of constraints and solvers.

Conclusion and Related Work

We have presented an implementation of the meta-solver framework as described in (Hofstedt 2001). The meta-solver coordinates the cooperative work of arbitrary pluggable constraint solvers. Redesign of an early proof-of-concept prototype now provides better modularization and encapsulation of the termination conditions. To cushion the inherent collaboration overhead and resulting performance problems

```
(define-strategy heuristic-solver-flow
    (heuristic-strategy)
  (:step
   (select ((eq-constraints (= t t))
            (in-constraints (in t t))
            (rest t))
     (tell-all in-constraints)
     (tell-all eq-constraints)
     (tell-all rest)
     (project-one my-linar)
     (tell-all)
     (project-all))))
```

Figure 2: Strategy specification for the heuristic-solver-flow strategy.

of meta-solver systems stemming from the necessary communication between the participating solvers, we designed a strategy language that enables the definition of problem domain tailored cooperation strategies. With the help of a positional pattern language and taking advantage of the encapsulated termination conditions it is possible to easily enhance or alter existing basic strategies for problem specific needs. The effect of such strategies has been verified on several multi-domain constraint problems.

In recent years several approaches for cooperating solvers have been presented (Kobayashi *et al.* 2002; Monfroy 1996; Rueher 1995). In the works of Monfroy and Rueher cooperation is based on a fixed set of strategies and solvers. The architecture of Kobayashi *et al.* is quite similar to the one of Hofstedt. However, with the presented strategy language our system gives finer control over the individual collaboration steps than the abstract control facilities employed in other systems. The level of abstraction used depends on the desired strategy. To our knowledge, the grouping of conjunction parts based on their structure as exercised by our pattern facility has not been applied in any similar system.

References

Bitner, J., and Reingold, E. M. 1975. Backtrack Programming Techniques. *Communications of the ACM (CACM)* 18(11):651–655.

Hofstedt, P.; Seifert, D.; and Godehardt, E. 2001. A Framework for Cooperating Constraint Solvers – A Prototypic Implementation. In *CoSolv Workshop, CP'2001*.

Hofstedt, P. 2001. *Cooperation and Coordination of Constraint Solvers*. Ph.D. Dissertation, Technische Universität Dresden. Shaker Verlag, Aachen.

Kobayashi, N.; Marin, M.; Ida, T.; and Che, Z. 2002. Open CFLP: An Open System for Collaborative Constraint Functional Logic Programming. In *WFLP'2002*.

Monfroy, E. 1996. *Solver Collaboration for Constraint Logic Programming*. Ph.D. Dissertation, Université Henri Poincaré – Nancy I.

Rueher, M. 1995. An Architecture for Cooperating Constraint Solvers on Reals. In Podelski, A., ed., *Constraint Programming: basics and trends*, volume 910 of *LNCS*, 231–250. Springer-Verlag.