A flexible Meta-Solver Framework for Constraint Solver Collaboration

Stephan Frank[1], Petra Hofstedt[1], and Pierre R. Mai[2]

[1] Berlin University of Technology
{sfrank, ph}@cs.tu-berlin.de
[2] PMSF IT Consulting
pmai@pmsf.de

Abstract. The solving of multi-domain constraint problems with the help of collaborating solvers has seen extended interest in recent years. We describe the implementation (Meta-S) and extension of a previously proposed theoretical framework of cooperating constraint solvers. Meta-S allows the dynamic integration of arbitrary external (stand-alone) solvers to enable the collaborative processing of constraints. The modular structure allows the easy experimentation with different cooperation strategies. This is further amplified by an integrated strategy description language which supports the adaption of general strategies for problem specific needs to reduce the incurred cooperation overhead.

1 Introduction

Constraint solving has seen active research and application during the last decades. The computational costs were reduced by the introduction of advanced heuristics and sophisticated solving algorithms. However, constraint solvers are limited to restricted domains for efficient processing. Many interesting problems, on the other hand, are intuitively expressed as multi-domain descriptions. The effort for developing new constraint solvers that are able to handle such multi-domain problems is significant. A different approach, researched actively during the last years, is the employment of meta-solvers that handle the overall solving process by managing the collaboration of a number of individual (preexisting) constraint solvers.

The theoretical meta-solver framework developed by Hofstedt [1] provides a sound architecture for the cooperation of arbitrary constraint solvers. In this approach mixed-domain constraint problems are split into their single-domain subproblems which can be tackled by the individual solvers. Such multi-domain problems can then be solved cooperatively while none of the individual participating solvers would be able to handle the problem on their own.

The system architecture consists of a global pool associated with the meta-solver and any number of pluggable arbitrary (external) constraint solvers. Initially the pool contains a constraint conjunction which is to be solved. The meta-solver then takes constraints of this conjunction and propagates them to the individual participating solvers. This is done via the interface function *tell*.

The solvers put these propagated constraints into their stores and derive new information (i.e. new constraints). These newly derived constraints are then in return requested by the meta-solver via the function *project* in order to add them to the global pool. This *tell* – *project* cycle continues until either a failure occurs, i.e. the constraint problem is unsatisfiable, or the pool eventually is emptied, i.e. no contradiction could be found.

A previous proof of concept implementation [2] worked quite satisfactorily. However, due to the monolithic structure it was difficult to experiment with changing solver collaboration and constraint propagation strategies to decrease the inherent communication overhead of the meta-solver approach. Taking aboard the lessons learned we developed an enhanced and revised framework which provides the ability to easily define generic and problem-specific solving strategies. This task was well supported by the dynamic features of the Common Lisp (CL) programming language.

The remainder of the paper is organized as follows. The next section introduces related work of the constraint community. After that, Sect. 3 describes the overall system architecture thereby focusing on internal details necessary for later explanations. This is followed by the illustration of an actual constraint problem definition in Sect. 4. General solving strategies influence the behavior of the overall meta-solving process. Section 5 focuses on the development of such strategies with special emphasis on the termination conditions and general approaches. This part is completed by the introduction of a specialized strategy specification language in Sect. 5.3 which supports the user to easily express more sophisticated algorithms. Finally, Sect. 6 focuses on the problem of value conversion to ensure broader information flow. The paper is concluded in Sect. 7.

2 Related Work

Active research within the field of constraint programming during the last decades has yielded many interesting directions of sophisticated solving algorithms as well as approaches of collaborating solvers to join the capabilities of different solvers. Hong [3] was one of the first to suggest the basic scheme cooperative solvers operate on. Each individual subsolver is repeatedly applied until a fixed-point is reached and hence no further changes occur. Hong especially focuses on the confluence of his cooperation system. The termination and confluence suppositions for Meta-S are extensively elaborated in [4].

Rueher [5] presents an agent-based concurrent cooperation scheme with a fixed set of collaborating solvers and only a single predefined cooperation strategy. All solver functions are defined on the same type (particularly real numbers).

Like our architecture the BALI environment for executing constraint solver combinations by Monfroy [6] provides a glass box mechanism to link a (predefined) set of (heterogeneous) black box solvers. BALI provides a fixed set of three cooperation primitives: sequential, concurrent and parallel; it is integrated into a logic host language. Formulation of BALI's cooperation operations within our framework is discussed in [4]. Meta-S provides a more fine-grained control

for strategy development. New cooperation strategies can be easily defined and plugged in, just like additional solvers (cf. Sect. 5). Different host languages can be integrated into the framework by treating them as constraint solvers [7]. In [8] Castro and Monfroy extend the ideas of BALI. Compared to Meta-S the available fixed strategy-operator set is more abstract though allows a similar strategy description. Our explicit distinction of propagation and projection phases enables a more fine-grained strategy definition and optimization. The strategy operators of [8] could be provided as predefined core functions in our system.

OpenCFLP by Kobayashi *et al.* [9] is a system for solving *equations* by collaborating solvers. It allows the plug-in like integration of arbitrary solvers in conjunction with a declarative strategy definition basing upon a set of basic operators. However, we believe the presented strategy language of Meta-S combined with the structural pattern-matching and optional rewriting facilities provide finer and more intuitive control for strategy development.

3 Architecture

Since Meta-S was intended to serve as a test-bed for cooperation strategy experimentation the whole architecture was designed in a highly modular way to allow even radical changes without interfering with other parts of the framework. As illustrated in Fig. 1 *Base* and *Meta* are the two main modules representing the meta-solver system, together with any number of pluggable constraint solvers. The special role of the CL-solver will be illustrated in Sect. 6. Note that Fig. 1 only depicts the overall module structure, not the class structure as some CL Object System features are not easily describable graphically (e.g. in UML).

3.1 System Decomposition

The Base module contains the basic data structures and fundamental functionality of the framework. This includes the external and internal representations of constraints and the ability to translate between those representations.

A pattern language provides positional pattern expressions for matching constraints depending on their structure and contents. Combined with a template mechanism this enables (sub)constraint selection and rewriting of constraints.

Here also resides the abstract interface providing the connection between the meta-solver framework and the attached solvers. Using this solver interface we have connected a number of constraint solvers from several domains: a solver for linear arithmetic, finite domain (FD) solvers for different domain types (floats, strings and rationals), and an interval arithmetic solver.

The *Meta* module contains the meta-solver proper and additional user facilities to support the command-line interface as well as programming constructs for the integration within other applications. Furthermore, here the strategy framework for basic and enhanced strategy development resides.

Instances of the class *meta-solver*, which is at the core of the module, represent each an actual meta-solver system consisting of an initial constraint disjunction branch (usually only a conjunction is given initially), defined variables with

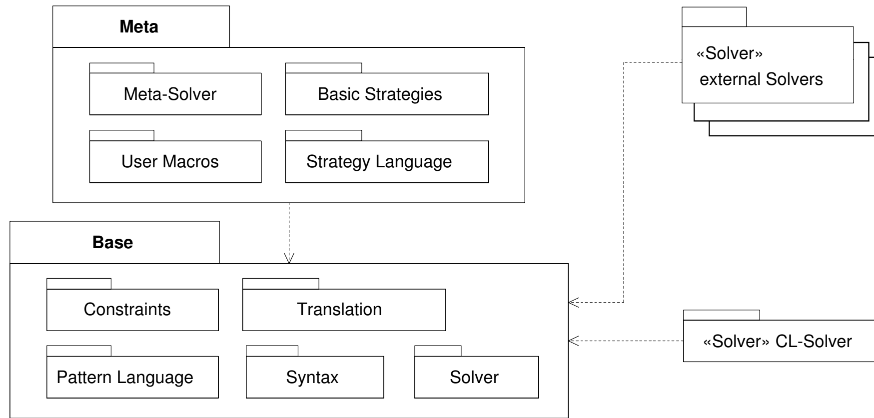

Fig. 1. The overall module structure of the Meta-S framework

associated types and the participating solvers. They each represent a complete constraint problem with associated state and solver information that is to be solved or simplified.

3.2 A Closer Look Inside

In [4] a *configuration* is defined as the state of the architecture for one constraint conjunction. Such configurations are cloned when a disjunction is encountered during projection – one copy for each disjunction part. The *meta-solver* class keeps track of state information common to all configurations whereas another class *meta-config* corresponds to the above configurations where each instance represents a possible solution set, similar to the branches of a search-tree. State information of a configuration covers the set of pending constraints (the constraint pool), the individual (external) solver states and strategy-specific information like pending disjunctions and the current projection mode (see below).

The state information for each solver is tracked by instances of the *solver-config* class which are referenced by the according *meta-config* states. The *solver-config* class implements the minimal interface to enable control and access of external solvers by the superior meta-solver. The abstract solver interface consists of a number of generic functions (GFs) that need to be realized by an actual solver implementation/bridge:

- GFs for the creation, initialization, cloning and destruction of configurations. The ability to clone the solver states (including its associated store) is required to implement backtracking along different branches of a disjunction.
- Two GFs for propagation of constraint conjunctions and projection of disjunctions of constraint conjunctions respectively.

Similar to *solver-config*, the class *meta-config* offers the interface to control the different configurations being handled by the meta-solver during the solving process. The interface functionality consists of the following areas:

– GFs for life-time control of configurations, i.e. creation of initial configurations, cloning upon encountering disjunctions for backtracking purposes, and destruction once a configuration has been declared no longer needed due to a detected inconsistency in the constraint problem for this branch.
– GFs for the addition of new constraints, or more specifically (disjunctions of) conjunctions of constraints, to the constraint pool of a configuration for further propagation to the external solvers. This function signals success or failure (if the newly added constraint enabled the solver to derive an inconsistency). External solvers are free to delay the consistency checking until the projection phase.[3]
– One generic function (*meta-config-run*) that implements the solving process for a particular configuration. This function will be invoked by the meta-solver and continues until the configuration is solved or declared unsatisfiable by any of the cooperating solvers. In the latter case the configuration is destroyed, otherwise "success" is signaled.
  Furthermore it is possible to suspend the solving process of a configuration, yielding control back to the meta-solver. This might be useful to limit processing depth/time per configuration for certain strategies, especially when the task is not to find all but only one solution. Suspended configurations can be restarted again at any time.
– A GF is provided to extract the final constraints and variable bindings of a successfully solved configuration for further processing and presentation.

One optimization we included is the delayed instantiation of disjunction branches since heavy cloning can lead to quite dramatic space requirements. This obvious *disjunction parking* space optimization technique creates only one additional copy of the current configuration and continues to use this newly cloned configuration for one single part of the disjunction. The other backup copy is used to spawn additional clones step by step when the remaining, parked disjunction branches are processed. Since earlier branches and their configurations may have already failed and are consequently destroyed this can lead to noticeable memory savings.

A second, optional optimization to reduce the number of life configurations taken over from [2] is the distinction of two different projection categories: *weak* and *strong projection* which can be easily integrated into the solver connection interface. Weak projection tries to reduce the number of disjunctions by rewriting constraints of the form $X = 1 \vee X = 2 \vee X = 3$ to the disequations $X \geq 1 \wedge X \leq 3$ whereas strong projection simply returns the unaltered disjunction, i.e. weak

[3] This also enables solvers to delay constraints internally. Such techniques are often used in linear arithmetic solvers to process nonlinear constraints – these constraints are delayed until enough variables are bound to exact values such that the constraint becomes linear and can be processed eventually.

projection always returns conjunctions only, whereas strong projection can also yield disjunctions. Incorporation of a weak projection phase thus delays (and potentially reduces) the occurence of constraint disjunctions. Weak projection is optional and it is obviously necessary to ensure that a strong projection phase is always initiated eventually (cf. the discussion of termination in Sect. 5.1).

Before going into detail how the described interface functions can be applied and enhanced for (meta-)solving strategy development, we will shed some light on the actual definition of constraint problems.

4 Defining Constraint Problems

Since we were using Common Lisp as the implementation language it was deemed sensible to use an s-expression[4] based input syntax as basis for our constraint problem definition language. While the reservoir of defined operators and relations that are usable as part of the problem specification is *only* dependent on (and dynamically inherited from) the applied/currently connected constraint solvers, using s-expressions, which are fully parenthesized, had the additionally benefit of avoiding any issues with operator precedence.

However the simple nature of s-expressions might not be appropriate for certain domains. Hence we provide a syntax extension facility, based on CL reader macros, that allows the definition of specialized (sub)syntaxes for the input description, while retaining the benefits of s-expressions. An example for one such syntax-extension would be the introduction of an optional infix-notation for mathematical expressions. The set-notation for the domain elements (using braces – {}) in the example in Fig. 2 (lines 9–16) is another application of extended syntax. Here, the underlying general s-expression syntax would require nested calls to the `add-to-domain` and `empty-domain` constructor operators provided by the FD-solver, because of the fixed arity of constructors in Meta-S.

Figure 2 displays a (simple) complete example of a (row-wise) definition of the well-known SEND-MORE-MONEY crypto-arithmetic problem. An equivalent column-wise description is also possible, where the columns are added to temporary variables which are then multiplied by a ten-power factor depending on their position. Line 1 initiates the loading of the finite domain specific syntax extensions. Line 3 starts the problem description assigning it the name `*smm-row*`. The meta-solver will use the *eager* strategy (line 4, cf. Sect. 5) with the given preferred variable ordering for projections. Lines 6–7 state the used solvers (`fd-rational`, a FD-solver on rational numbers and `cllin` – a linear-arithmetic solver) and assign names to the solver instances to enable access by name later on. The following line 8 declares constraint variables of type `rational`. The remainder of the example specifies the actual constraint conjunction for the SEND-MORE-MONEY problem: First the possible domain values (lines 9–16), the demanded inequalities between all one-letter variables (line 17), and finally the sums to compute the values for the variables `send`, `more`, and

[4] s-expressions are the textual presentation of complex tree-structures used e.g. by Lisp-like languages

```
 1 (with-file-syntax :finite-domain)        ;; load domain specific syntax
 2
 3 (define-meta-solver *smm-row*            ;; problem definition
 4     (meta-solver eager-strategy          ;; select the eager strategy
 5                 :variable-ordering '(send more money m o n e y r s d))
 6   ((my-fd-rational fd-rational)          ;; employ a FD and a linear arith-
 7    (my-linar cllin))                     ;; metic solver
 8   ((rational s e n d m o r y send more money)) ;; declare variable types
 9   (((in s #{0 1 2 3 4 5 6 7 8 9})        ;; define the problem conjunction
10     (in e #{0 1 2 3 4 5 6 7 8 9})
11     (in n #{0 1 2 3 4 5 6 7 8 9})        ;; in and all-different are provi-
12     (in d #{0 1 2 3 4 5 6 7 8 9})        ;; ded by the FD-Solver, = by both,
13     (in m #{0 1 2 3 4 5 6 7 8 9})        ;; and the +, * operators by
14     (in o #{0 1 2 3 4 5 6 7 8 9})        ;; our linear-arithmetic solver cllin
15     (in r #{0 1 2 3 4 5 6 7 8 9})
16     (in y #{0 1 2 3 4 5 6 7 8 9})
17     (all-different {s e n d m o r y})
18     ((= my-linar) send (+ (* (+ (* (+ (* s 10) e) 10) n) 10) d))
19     (= more (+ (* (+ (* (+ (* m 10) o) 10) r) 10) e))
20     (= money (+ (* (+ (* (+ (* (+ (* m 10) o) 10) n) 10) e) 10) y))
21     (= (+ send more) money))))
```

Fig. 2. Sample code for the Meta-S problem description of the SEND-MORE-MONEY variant with 25 solutions (i.e. leading zeros are allowed). It utilizes two cooperating constraint solvers: a finite-domain and a linear arithmetic solver.

`money` (lines 18–21). The possibility to refer to specific constraint solver instances is used in line 18 where the constraint relation '=' is annotated such that this constraint will only be propagated to the solver instance named `my-linar`. It enables *ad hoc* control of constraint propagation on the problem definition level. This mechanism is useful when several of the participating solvers overlap on the constraint/function capabilities but we know at the same time that one specific solver can handle certain input constraints comparatively better than the other collaborating solvers. It would be thus advantageous to restrict the constraint propagation to one or more specific solvers. Such concepts are of course also expressible within a specific strategy definition (cf. Sect. 5.3).

5 Strategy Development

The meta-solver based architecture causes some inherent communication overhead but provides the advantage of letting constraint solvers of different domains cooperate via the implementation of a simple interface. Solutions are derived as follows: Propagate constraints from the global pool to the external solvers. All solvers signaling a change are marked *dirty*. All dirty solvers are then requested to project their information. These constraints are then again added to the pool. This cycle ends when the pool is emptied or a solver signals a conflict which marks the current setting unsatisfiable. The flow of information (i.e. newly

derived constraints and assignments) between the participating solvers greatly influences the overall efficiency. Unfortunately, an effective cooperation strategy depends on the problem (and its modeling) as well as on the number and capabilities of the cooperating solvers and can thus not be generally formulated.

Meta-S provides the user and strategy developer with facilities to develop solving strategies within different levels of abstraction and detail.

The class *meta-config* (cf. Sect. 3.2) is the natural place to implement different solving strategies by providing complete implementations of the generic function *meta-config-run*, each representing a different solving strategy. However, using this approach much code would be duplicated by the different strategy implementations. Recognition of this fact led to the introduction of the abstract *strategy* class which provides a more specific interface and default implementations for a number of generic functions, thus allowing the developer of strategies to override default code at a much finer granularity.

5.1 Termination Conditions

For the creation of the strategy class it was vital to encapsulate the essential termination conditions which can be traced independently and without particular assistance by the strategy proper. Thus the strategy developer can concentrate on the information flow only, without having to keep track of the termination conditions explicitly.

The state information of a configuration as managed by *meta-config* like described in Sect. 3.2 can be used to formulate the termination criteria which correspond to the conditions described and proven in [4]. Termination of the solving process of a particular configuration eventually ensues when:

1. there are no pending constraints left in the pool, i.e. all constraints have been propagated to the external solvers,
2. there are no parked disjunction left,
3. none of the solvers is marked *dirty*, i.e. there are no changes that have not been requested yet by projection,
4. the current projection mode is *strong projection*.

Tests for these criteria can be run independently and are easily outsourced from the solving strategy itself. Updating the individual criteria states is done when the actual actions changing them are performed. We can thus describe strategies without explicit checks for termination and without extra update code for the termination states. Consequently any strategy that takes care to eventually empty the pool, project all propagated variables and switch to strong projection will terminate and not exit prematurely. Otherwise a strategy is essentially free to do whatever it wants by using the full power of Common Lisp. It is of course still possible to write non-terminating strategies, e.g. by implementing endless loops or by never propagating any constraints from the pool.

The solving process for a configuration now essentially consists of the following actions:

1. Check if there are pending constraints and/or dirty solvers. If "not", then we are finished, otherwise
2. Invoke *strategy-step* (cf. Sect. 5.2) for a single solving step where constraint propagation and projection to/from the external solvers is handled.
3. Repeat step 1, if *strategy-finish* does not succeed, i.e. not all termination conditions are met.

strategy-finish provides expanded termination checking since strategies may employ optimizations that delay or suppress the addition of constraints to the global pool, like e.g. the above mentioned disjunction parking technique. The time needed to meet the termination criteria is highly dependent on the overall collaboration method. Providing efficient collaboration methods is vital for acceptable performance. This problem will be addressed in the next sections.

5.2 General Strategies

To provide easy access points for strategy development the *strategy* class extends the interface of *meta-config* with fine-grained action-oriented hooks where the user can add and replace code by deriving new classes. We provide default implementations for these generic interface functions that form the base for all strategies. Interface functions are divided into the following categories:

- delaying and undelaying of constraints,
- constraint propagation and projection of individual or all (dirty) solvers,
- rewriting and conversion of projected constraints from the originating solver to other solvers (cf. Sect. 6), and
- additional termination testing (*strategy-finish*) and restart actions when awaking suspended configurations (*strategy-restart-actions*).
- The function *strategy-run* that constitutes the *overall* control loop of the whole strategy. It installs the hooks to allow dynamic exits and terminates by indicating success, failure or suspension, depending on the solver return states and strategy actions. It is invoked by the meta-solver action loop.
- *strategy-step* represents the processing of a single control loop run and is thus repeatedly invoked. It is the most natural place for strategies to establish variations of ordering, classification or form of propagation and projection actions (cf. Sect. 5.3).

In order to utilize the flexibility of the abstract *strategy* class, three generic strategies were implemented. They are all realized by defining replacements of the default methods for disjunction adding and for extended termination testing (in *strategy-finish*), thus the main strategy loop was left unaltered. All strategies could be realized in under 50 lines of code combined. The implemented strategies currently all emphasize on the handling of returned disjunction within the projection phase:

eager This strategy propagates all constraint information as early as possible in the solving process. Returned disjunctions (during projection) lead to

immediate spawning of cloned configurations for the individual disjunction branches. The partition into two projection stages (weak and strong projections, cf. Sect 3.2) can be applied advantageously as it delays and potentially reduces the number of live configurations. The *parked disjunction* technique (cf. Sect. 3.2) helps to further reduce the memory requirements.

lazy This strategy is comparable to *eager* with the major difference that the backtrack-branches of disjunctions (returned by projections) are not created immediately but possibly remaining constraint conjunctions are propagated first to the solvers. This is done in the hope that it will help to reduce the lifetime of dead-end backtracking branches. Once all conjunctions are processed the cloning actions for the disjunction branches are initiated. The deferred handling of disjunction obviously clashes with our weak/strong-projection partitioning scheme. For solver plug-ins that adhere the optional *weak* option, this strategy is thus inferior to *eager*. However, in cases where the (optional) weak projection phase is ignored by the solver interface, this strategy might prove advantageous over our eager strategy.

heuristic This last strategy integrates the previous two and additionally implements a heuristic element commonly known as *fail first* [10]. Here, the disjunction that is most likely to lead to an inconsistency (i.e. the one with the least branches) is processed first to keep the backtracking graph lean. This comes at the additional cost of more strong projections since disjunctions are requested from the solvers to decide which one (disjunction) to choose next, depending on the number of branches. After the strong projection round, the disjunction with the least number of elements is used for processing; the remaining disjunctions are discarded as their validity might be already obsolete at a later time. However, despite the greater number of strong projections it turns out that this strategy provides better results for most of our example problems.

The implemented example strategies have been tested with several different constraint problems exercising variations of constraint solver cooperation sets. Results for the SEND-MORE-MONEY problem runs behave representatively and are displayed in the upper half of Table 1. The strategies provide an optional parameter to allow the definition of a variable order that should be used for projection. This consequently also influences the order for the propagation of derived constraints and thus is essential for the overall runtime. Rows tagged *ordered* are the runs where we provided the strategies with a beneficial variable order whereas the other rows operated on an arbitrary projection order.[5]

The table emphasizes the important role of the projection order and shows that the heuristic strategy is able to derive a good order without the need for additional user supplied information. This is important as such an order is usually not known *a priori* nor easily derivable.

[5] Care was taken that the orders were identical for the different strategy runs.

strategy	time in s	# of clones	# of propagates	# of projections weak	strong
`lazy`	79.0	3521	44918	18909	16126
`lazy`, ordered	27.7	5269	36539	2761	25366
`eager`	91.2	2502	49891	20933	2068
`eager`, ordered	9.2	253	6368	2717	858
`heuristic`	10.2	73	6097	3829	1276
`heuristic`, ordered	8.4	45	4850	2915	1012
`domain-eq-first`	9.1	253	6149	2728	858
`once-domain-eq-first`	9.2	253	6149	2728	858
`solver-flow`	6.3	253	4763	2013	858
`heuristic-solver-flow`	5.4	45	3808	2123	1012

Table 1. Benchmarking results for generic and problem-specific strategies for the SEND-MORE-MONEY crypto-arithmetic puzzle (column-wise definition; 25 solutions, i.e. leading zeros are allowed)

5.3 Problem-specific Strategies

It is obvious that control of information flow can gain important performance increases. Despite the comfortable performance of the different base strategies (`eager`, `lazy` and `heuristic`) it should be possible to influence strategy actions at a higher level. Such a feature allows the incorporation of problem and domain specific knowledge that would help speeding up the solving process. To support the user in this important task we developed a strategy language that allows to devise new more efficient strategies by extending generic strategies with additional control flow specifications.

Figure 3 presents one example strategy called `heuristic-solver-flow`. It was derived by extending the general heuristic strategy with a more optimized information flow. The :`step` clause initiates the inner strategy control loop definition. We start by partitioning all the constraints of the global pool into three groups based on their structure using the pattern matching facility (lines 4–6):

`eq-constraints` are constraints with an equality relation (`=`),
`in-constraints` match constraints with the `in` relation,
`rest` match all other constraints. This is expressed by the wildcard `t`.

The classified constraints are then propagated to the external solvers in the order of the groups given in lines 7–9. This is supported by our knowledge that this order helps the FD-solver and the linear arithmetic solver to populate their internal data structures more effectively. In the next step the linear arithmetic solver instance `my-linar` is requested to project its knowledge (line 10). The received constraints are again removed from the pool and propagated to the other solvers (line 11). Finally, the remaining solvers are requested to project their derived constraints (line 12). This solving step is repeatedly initiated by the controlling strategy until the termination conditions are satisfied.

```
1  (define-strategy heuristic-solver-flow
2     (heuristic-strategy)                ;; use heuristic as base strategy
3    (:step
4     (select ((eq-constraints (= t t))   ;; patterns for eq/in constraints
5              (in-constraints (in t t))
6              (rest t))
7       (tell-all in-constraints)         ;; first propagate in
8       (tell-all eq-constraints)         ;; and eq constraints
9       (tell-all rest)                   ;; then all others
10      (project-one my-linar)            ;; project from my-linar
11      (tell-all)                        ;; and propagate, then
12      (project-all))))                  ;; project from all other solvers
```

Fig. 3. Strategy specification for the heuristic-solver-flow strategy

The lower part of Table 1 shows the performance data for four different specific strategies. The individual strategies all prioritize different types of constraints and/or different solver instances and clearly show the advantage of problem-tailored strategies for our setting.

The design of our strategy language anticipates that the user wants to control the exact order of constraint propagations, basing on different criteria like the target solver and/or the structure/properties of constraints. Additionally optional rewriting of returned constraints and influence on the projection sequencing should be possible.

Founded on these considerations the strategy language provides the following set of constructs and properties:

- We provide a set of support functions and macros, but the language is not limited to them. Due to the integration into the CL environment, all normal Common Lisp constructs can be applied.
- The execution of strategies is at the core of the meta-solving process and should thus run with high efficiency. This is ensured by automatically compiling the strategy language (on invocation) to native code (via CL). Therefore no interpretation overhead is carried into the inner execution loop.
- The integrated pattern language allows the classification/selection, destructuring (i.e. splitting into sub-parts matching specified patterns) and rewriting of constraints based on their structure by using positional pattern expressions. Destructuring of constraints is supported in pattern expressions by binding local (strategy) variables to matched subexpressions. Partitioning a set of constraints (like the set of pending constraints in the global pool) is possible as well. Creation of new constraints within the rewriting process is supported by a template mechanism.
  The matching expressions of the pattern language are compiled to native code, like the whole strategy language.
- Primitives for propagation of sets of constraints, as well as projection of specific or all solvers against a given set of variables and terms, support a detailed information flow control.

6 Conversions and the CL-Solver

Since we are working on different domains and solver types, the underlying relations, operators and types may be different. The meta-solver will split initial problems such that every participating solver only receives subproblems it is able to manage. However, to ensure efficient processing of the overall constraint problem, additional information flow between different solver types is desirable. Hofstedt [4] introduces the function $conv_{\nu \to \mu}$ which converts disjunctions of constraint conjunctions between the languages of two solvers ν and μ.

Such conversions are present in Meta-S at two different places. First, conversion of constraint conjunctions and disjunctions can be seen as a way of rewriting. Such rewriting can be applied as an intermittent step in the projection phase, using the built-in rewriting capabilities. Here, the solver-returned constraints are rewritten, before they are seen by the meta-solver and before being added to the global pool. Only the then rewritten constraints are considered for further processing. Another possibility is the application of an external rewriting solver, e.g. one implementing CHR [11]. This solver will participate in the overall problem solving process like any other cooperating solver. The original constraint conjunction of appropriate type is propagated to the rewriting solver and then the rewritten constraints are returned during the projection phase.

A second type of interesting conversion common to many (multi-domain) applications, is the transformation between disjoint though still fairly compatible value types. Consider two solvers, one operating on floating point numbers, the other on rationals (like our interval and linear arithmetic solvers). Since these number types are disjoint, ordinarily no information interchange (like value bindings) would happen between the two solvers. The CL-solver offers a residuation [12] based approach to define conversion operators using the functions of the underlying CL system. Figure 4 shows the extract of a constraint problem definition that defines a CL-solver instance `my-conversions` with two operators `to-dfloat` and `to-rat` which convert from double-floats to rationals and vice versa respectively using the standard Common Lisp functions `coerce` and `rationalize`. Additionally the appropriate equality relations are declared. These ensure that the CL-solver receives all bindings and can thus react on all variables captured in `to-dfloat` and `to-rat` applications. Note that such conversions are not necessary for existing subtype relations, like e.g. within the standard numeric tower of CL, where information flow will be propagated automatically by the meta-solver in the direction of the subtype relation because there is no type mismatch.

Of course this approach is fraught with problems when the coercions involved are imprecise, i.e. result in loss of information, which is the case with the numeric types mentioned. Invoking such conversions several times during a meta-solver run can quickly lead to a significant information loss. This is all the more problematic, because common techniques for analyzing the numerical accuracy of algorithms are hard to apply in the setting of a meta-solver system, where the overall algorithm is fairly involved and dynamic in nature, making the exact flow of numeric information hard to discover. We are convinced that further investigation of the general problem of conversion between representations is needed.

```
1 (my-conversions cl-solver
2                 :operators '((to-dfloat (rational) double-float
3                               (lambda (x) (coerce x 'double-float)))
4                              (to-rat (double-float) rational rationalize))
5                 :relations '((= (rational rational) =)
6                              (= (double-float double-float) =)))
```

Fig. 4. Definition of a conversion solver instance `my-conversion` for converting *double-float* to *rational* numbers (and *vice versa*) by applying the CL-solver

Nevertheless, putting aside the numeric problems while converting numbers, the easy conversion of more abstract data types (when possible) is still an interesting option.

Even though the CL-solver works on the principle of residuation and thus represents a quite strong barrier of information flow, conversion worked surprisingly well in our experiments. It additionally provides a simple way to integrate the meta-solver system within larger Common Lisp applications.

The ability to operate on arbitrary CL functions provides the additional benefit of simple Meta-S integration into larger applications. Variable bindings for dynamically created or statically described constraint systems are established by simple function calls within the CL-solver when a variable value is requested (which can e.g. issue GUI input actions by the user).

A possible option to reduce the information flow barrier posed by the residuation principle used by the CL-solver would be the application of a specific, more intelligent conversion solver. Especially the knowledge about possible inverse relations and operators would provide a more flexible information propagation.

7 Conclusion

The research described in this paper was motivated to provide a usable implementation of the theoretical framework by Hofstedt [1, 4]. Integrating the lessons learned in a previous proof of concept implementation [2], one of the goals was to provide a flexible architecture to allow the easy exchange of core modules without interfering the rest of the system. This supports experimentation with different strategy approaches to cushion the inherent collaboration overhead and resulting performance problems of meta-solver systems originating from the necessary communication between the participating solvers (and the meta system).

The paper started with a description of the overall module structure (Sect. 3), highlighting implementation details where necessary. The notion of *configurations* and their resemblance in the implementation as meta/solver-configs was introduced to hold the state of a computation branch thus allowing backtracking. Section 4 continued with an illustration of the problem definition facilities.

Section 5 opens the strategy part of the paper. By encapsulating the termination conditions we were able to ease the task of implementing different collaboration strategies since state and termination condition tracking could be

outsourced to a lower part of the system and maintained automatically. We continued with the development of three different cooperation strategies, each focusing on a different part of the overall information flow (Sect. 5.2). Observing that propagation and projection orders are essential for acceptable performance and appropriate heuristics vary for different problems and settings, we introduced a strategy description language in Sect. 5.3. With the help of a positional pattern language it was possible to easily alter the core solving steps of existing strategies for problem specific needs. The effect has been verified on a number of multi-domain constraint problems. We believe that current performance barriers are imposed by (speed) limitations of the plugged-in solvers (particularly the linear-arithmetic solver). A future work will be the integration of more sophisticated solvers and the study of the effects of a stronger multi-domain mix.

Finally, Sect. 6 discussed the problem of constraint value conversion to support a stronger information flow. Such handling is quite delicate for numerical issues though necessary to provide a stronger information interchange between the individual solvers. It is certainly useful for non-numeric values. However, more experimentation is needed in this area to further reduce barriers for value/constraint propagation (of disjoint types).

References

1. Hofstedt, P.: Better Communication for Tighter Cooperation. In Lloyd, J., ed.: First Int'l Conf. on Computational Logic – CL'00. Volume 1861 of LNCS. (2000)
2. Godehardt, E., Hofstedt, P., Seifert, D.: A Framework for Cooperating Constraint Solvers. In: CoSolv Workshop. At the 7th Int'l Conf. on Principles and Practice of Constraint Programming. (2001)
3. Hong, H.: Confluency of Cooperative Constraint Solvers. Technical Report 94-08, Research Institute for Symbolic Computation, Linz, Austria (1994)
4. Hofstedt, P.: Cooperation and Coordination of Constraint Solvers. PhD thesis, Technische Universität Dresden (2001) Shaker Verlag, Aachen.
5. Rueher, M.: An Architecture for Cooperating Constraint Solvers on Reals. In Podelski, A., ed.: Constraint Programming: basics and trends. Châtillon Spring School 1994. selected papers. Volume 910 of LNCS. (1995)
6. Monfroy, E.: Solver Collaboration for Constraint Logic Programming. PhD thesis, Université Henri Poincaré – Nancy I (1996)
7. Hofstedt, P.: A general Approach for Building Constraint Languages. In McKay, B., Slaney, J., eds.: AI 2002: Advances in AI. Volume 2557 of LNCS. (2002)
8. Castro, C., Monfroy, E.: Basic Operators for Solving Constraints via Collaboration of Solvers. In: Proceedings of AISC. Volume 1930 of LNAI., Springer-Verlag (2000)
9. Kobayashi, N., Marin, M., Ida, T., Che, Z.: Open CFLP: An Open System for Collaborative Constraint Functional Logic Programming. In: 11th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2002). (2002)
10. Bitner, J., Reingold, E.M.: Backtrack Programming Techniques. Communications of the ACM (CACM) **18** (1975)
11. Frühwirth, T.: Constraint Handling Rules. In Podelski, A., ed.: Constraint Programming: Basics and Trends. Volume 910 of LNCS. (1995)
12. Aït-Kaci, H., Nasr, R.: Integrating logic and functional programming. Lisp and Symbolic Computation **2** (1989)