A Master-Slave Architecture to Integrate Sets and Finite Domains in Java

F. Bergenti[1], E. Panegai[2] and G. Rossi[2]

[1] Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
bergenti@ce.unipr.it
[2] Dipartimento di Matematica
Università degli Studi di Parma
panegai@cs.unipr.it, gianfranco.rossi@unipr.it

Abstract. This paper summarizes the lessons learned from the integration of two Java constraint solvers: a set solver (namely JSetL) and a finite domains solver (namely JFD). The most relevant outcome of this experience is the definition of a generic master-slave architecture that can be used to support the cooperation of different solvers. Each slave is responsible for managing constraints of a particular sort and the master, which is also a solver, is in charge of distributing tasks according to a static, a-priori policy. This paper first presents this generic architecture in an abstract form; then, its concrete instantiation to the selected case study, i.e., the integration of JSetL and JFD, is also described. This case study was selected because it fully demonstrates the possibilities of this architecture as: *(i)* the poor performances of JSetL on non-set variables are overwhelmed by the cooperation with JFD; and *(ii)* the expressive power of JSetL is fully preserved and the integration with JFD demands no restrictions.

1 Introduction

Rarely a single constraint solver outperforms all others in all situations. As a matter of fact solvers are normally implemented on the basis of more or less explicit tradeoffs between:

1. Capabilities: the kinds of constraints they manage and under which assumptions; and
2. Performances: the adopted strategies, heuristics and optimizations.

If we accept the assumption that no single solver will be sufficient to accommodate the requirements of future applications, we should better take into account the possibility of synergically orchestrating the work of different solvers in order to efficiently overcome the limitations of each solver [9].

Along these lines, the main objective of this work is to implement and validate a case study of synergic cooperation between two solvers in the attempt to find out a generic approach that would help us in similar integration tasks. The realization of this combined solver guided us in the definition of a generic

master-slave architecture that can be easily and fruitfully applied in many interesting situations. The reason why we chose this bottom-up approach is because we believe that it can give us the instruments for deeply understanding and criticizing our results and, most notably, that it can ensure a fine-grained analysis of the impacts of important aspects, e.g., nondeterminism, in the cooperation of different solvers.

Many frameworks and architectures that explore the cooperative integration of constraint solvers are available in the literature (see, e.g., [6, 9]). They all share a common architectural outline made of the following three major components:

1. A top level of meta-resolution (namely a *meta-solver*);
2. A set of constraint solvers all grouped at an object level; and
3. An interface between the meta-solver and the object-level solvers.

Broadly speaking, our architecture can be thought as a contraction of this general architecture. The master-slave approach excludes the need of a layer of meta-resolution: one of the solvers is selected and promoted to the role of master. Such a solver does not work at the meta-level, rather it works at the object level and it is enriched with the capability of dispatching tasks to, and gathering results from, all other solvers. This may require some modifications to the selected master solver, but it saves us the trouble of implementing a solver from scratch when no meta-level functionality, e.g., expressing the dispatching policy in term of constraints, is requested.

The case study that motivated our work deals with the integration of two Java libraries JSetL [11] and JFD, both developed at the University of Parma[3], that implement respectively a constraint solver over sets and a constraint solver over finite integer domains. Both work well on their reference domains, but they are rather bad (or null) in all other situations: JSetL shows very poor performances on non-set variables; on the contrary, JFD shows good efficiency but it does not provide any notion of set or aggregate data type. Therefore, we decided to integrate the two solvers into an added-value Java constraint solver capable of exploiting:

1. The full expressive power of JSetL, with its inherent flexibility and generality; and
2. The efficiency of JFD in treating finite integer domains variables.

Integrating a constraint solver over general sets, namely CLP($\mathcal{SET}$) [5], and an efficient solver over finite domains, namely CLP($\mathcal{FD}$) [2], has been already explored in [4], where feasibility and usefulness of the approach are clearly pointed out. Since the constraint solving algorithms of JSetL and JFD are basically the same exploited in CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$), [4] also provides the theoretical basis of our current work. The work in [4], however, is deeply rooted in a logic programming framework, assuming a logic programming language as the common implementation language for the integrated solver. Moving to a more common programming context, such as that of Java, causes some implementation

[3] JSetL is available open source at `http://www.math.unipr.it/~gianfr/JSetL`. JFD is available on request from the authors of this paper.

decisions to become more evident—e.g., how to handle nondeterminism—and it requires explicit solutions for them to be provided. Moreover, differently from [4], we use the integration of the two specific solver as an opportunity to generalize the proposed solutions to a wider setting of cooperative constraint solving based on a master-slave architecture, possibly involving more than one slave solver.

The paper starts from the description of the generic master-slave architecture (next section) and its operational behaviour (Section 3). The instantiation of this architecture to our case study is postponed to Section 4. Then, our results are discussed together with other comparable results in Section 5. Section 6 draws some conclusion and presents some future work.

2 Master-Slave Constraint Solving

In this section we describe a generic master-slave architecture that we designed to support the integration of solvers with different capabilities. Each solver is responsible for a predefined set of constraints and the master, which is also a solver, is in charge of distributing tasks to, and gathering results from, the slaves. We work under the assumption that the distribution policy is static and based only on a fixed, a-priori mapping between kinds of constraints and solvers.

We start with a bunch of solvers to be integrated and we treat all of them, but the one selected to be the master, as black boxes. All solvers share a common high-level description as follows. We assume that each solver is equipped with a constraint store that holds its constraints. We also assume that all solvers regard each constraint as a collection of atomic constraints, possibly containing just a single atomic constraint, and interpreted logically as a conjunction of atomic constraints. We do not require all solvers to share the same set of atomic constraints. Typically, the sets of atomic constraints of the slaves are overlapping and some atomic constraint, e.g., the equality constraint, is in all sets.

As long as the interaction with the outer world is concerned, each solver is characterized in terms of:

1. The kinds of constraints that it can manage;
2. The way we can add constraints to its constraint store; and
3. The way the constraint store is made inspectable from the outside world in order to collect the results of computations.

The main task of each solver is to try to reduce any conjunction of atomic constraints to a simplified form that it cannot further simplify, i.e., that it considers *irreducible*. The detection of a failure (logically, the reduction to *false*) implies the unsatisfiability of the original constraint. Conversely, the ability to obtain the irreducible form may eventually imply the satisfiability of the original collection of constraints. In *complete* constraint solvers, the success of the reduction process allows to conclude the satisfiability of the original constraint and each of the obtained irreducible constraints represents a *solution* for the original constraint. On the contrary, *incomplete* constraint solvers usually return one irreducible constraint which has the same set of solutions as the original constraint, but

that is not guaranteed to be satisfiable (actually, determining its satisfiability can be potentially hard).

Our general master-slave architecture applies to both complete and incomplete solvers. However, the specific instantiation that we consider in our case study aims at developing a complete constraint solver in order to save the results of [4].

2.1 Selection of the Master

Given a set of solvers described in terms of the aforementioned characteristics, the first issue we have to tackle in order to enable cooperation among them regards assigning the role of master to a particular solver. The master represents the front end of the integrated solver and has the following duties:

1. It enables the programmer to express constraints;
2. It distributes tasks to slaves according to an allocation policy;
3. It gathers results from slaves; and
4. It provides a common view of results.

Differently from most of the approaches in the literature, our starting point is the selection of a solver to let it play the role of master. We are not concerned with realizing a new, special-purpose solver for the sole purpose of enabling cooperation. This would require the realization of a new solver from scratch and it would require to rewrite many algorithms and data structures that are already present in all solvers we are integrating. We would need a language for this new solver, a constraint store, and some rewriting procedures. All these features are already implemented and functioning in all solvers we are integrating.

Unfortunately, we cannot devise a single criterion for selecting the optimal solver to elect to the role of master. This choice is driven by the experience and by estimations of the required implementation effort. We can only enumerate some issues to consider for a good selection of the master:

1. The expressive power of the constraint languages involved, i.e., the master should provide a constraint language that subsumes the union of the languages of all slaves;
2. The performaces, i.e., slaves should perform better than the master in their reference domains;
3. The support for constraint programming abstractions, e.g., logical variables and nondeterminism;
4. The possibility of extending the selected solver to integrate master-specific procedures, e.g., a constraint dispatching procedure and its result processing counterpart; and
5. The public functionalities provided to programmers.

Reasonably, if we can access a solver with a good expressive power and such that it can cover most of the constraints of all other solvers, we have a good candidate for the role of master. Taking into account our case study, the constraint language of JSetL is sufficiently expressive and therefore we selected JSetL to play the role of master, after some minor modifications described in Section 4.

2.2 The Master's Solving Process

Once the role of master is assigned, the solving process works as follows. First, all constraints are loaded into the master's constraint store, then each constraint in the store is analyzed separately. For each constraint, the master performs one of the following mutually exclusive actions:

1. It solves the constraint with no help from slaves;
2. It delegates the resolution of the constraint to a bunch of slaves; or
3. It forwards the constraint to a bunch of slaves to let them exploit it for current or future use, and then it solves it.

The choice of which action to perform and when is a core part of the design of the master and it may heavily influence the overall speed-up that the integrated solver gains over single solvers.

Previous actions progress the master's constraint store towards a solved form and they are repeated until the solving process terminates successfully or until it fails. The process terminates successfully when no further constraint can be reduced or allocated to slaves and when all results are gathered from the constraint stores of slaves. On the contrary, it fails when the master, or any slave, detects an inconsistency and no further nondeterministic choices are left open.

2.3 Communications between the Master and the Slaves

In our architecture, the allocation of constraints to slaves is performed by means of a static, a-priori policy. This policy is chosen because it is very easy to implement and because it demands no considerable amount of computation compared to reasonable constraint solving processes.

Given an n-ary atomic constraint $C = op(t_1, \ldots, t_n)$, we first use the operator op and its arity n to allocate C to a group of solvers. If this is not sufficient to identify the needed solvers, we consider the data types of the t_i and we use this information to finally allocate C.

In general, given an atomic constraint C, and given S, the pre-computed set of slaves that can handle C, the allocation of C to slaves is performed by means of one of the four mutually exclusive cases:

1. If C is a constraint of the master and $S = \emptyset$, then the master solves it;
2. If C is not a constraint of the master and $S \neq \emptyset$, then C is posted to all slaves in S;
3. If C is a constraint of the master and $S \neq \emptyset$, then the master posts C to all slaves in S and then solves it; or
4. If C is not a constraint of the master and $S = \emptyset$, the allocation fails, i.e., the constraint is unknown.

The first option (that we call *MASTER*) is selected when the master can manage the constraint at hand and no slave needs it in for future computations. In our case study, this is the case of all set constraints, such as $op_{\subset}(x, y)$ which enforces $x \subset y$: JSetL reduces them with no help from JFD. The second option (called *SLAVE*) is the opposite, i.e., the master has no means to exploit the constraint and it needs a group of slaves to handle it. In our case study, this is the case of

arithmetic constraints like $op_+(z, x, y)$, which enforces $z = x + y$: JSetL cannot efficiently handle such a constraint and therefore it delegates it to JFD. The third option ($BOTH$) is selected when the master can handle the constraint on its own, but it knows that slaves may eventually need it. In our case study, this is the case of equality constraints like $op_=(x, y)$, which imposes $x = y$: if x and y are either free variables or integer constants, both JSetL and JFD need this constraint to make sure that no information contained in the original constraint store is lost during the solving process. The last of these four options ($UNKNOWN$) indicates an error in the integration of solvers and it should never occur.

The allocation of constraints to slaves is only the first part of the process the master performs to solve its constraint store. The second part consists of the master gathering the results from slaves and reconciling them into its constraint store. In our architecture, we do not take into account the possibility for a slave to generate constraints that are directly passed to other slaves.

The master is interested only in the final outcome of the computation of slaves, i.e., the irreducible constraints, involving variables known from the master, that the slaves leave in their constraint store after a complete reduction of any allocated constraint. Hence, the computation of slaves is immaterial from the point of view of the master.

Generally speaking, a slave should provide back to the master any constraint left in its constraint store that is available also in the language of the master. This would guarantee that no information is lost during the resolution and that each solver is free to choose the best approach for solving its constraints.

3 General Constraint Solving Procedures

The procedure the master performs for solving its constraint store is described in Algorithm 1, where *Store* is the conjunction of all constraints in the initial constraint store, conventionally ended with a *true*:

$$Store = C_1 \wedge C_2 \wedge \ldots \wedge true$$

This procedure moves from one atomic constraint C_i to the next C_{i+1} and solves each of them until *Store* is irreducible. The procedure *reset* sets the current constraint to point to the first atomic constraint in *Store*. Then, procedure *step* eventually allocates the current constraint to slaves or tries to solve it. Finally, function *is_final_form* tests if the reduction process is complete.

The master constraint solving procedure assumes that each slave provides a few methods to modify or inspect its constraint store, as follows.

3.1 Slave Interface

The following methods represent the interface that slave solvers provide to the outer world:

1. *add(C)*, where C is a slave constraint to be added to the constraint store of the slave;

Algorithm 1 The master's procedure for constraint solving

```
procedure master_solve(Constraint Store)
    repeat
        reset(Store);
        step(Store)
    until is_final_form(Store)
end procedure;
```

2. *get_constraints()* that returns the conjunction of constraints that are present in the store of the slave;
3. *solve(B)*, where B is a value from an enumerative type used to specify which solving process is requested.

The type of the argument of *solve* contains, at least, the value *REDUCE* and, possibly, the value *LABEL_ALL*. The first is used to ask the slave to process its constraint store until an irreducible form is found. The second asks the slave to label nondeterministically all variables in its constraint store. This process causes equality constraints $op_{=}(x, v)$ to be added to the constraint store of the slave to force $x = v$, where v is a value in the domain of variable x.

The previous methods are sufficient to let the master delegate constraints to slaves. However we need some glue functions to make a perfect match between the master's and each slave's view of the constraint store. Such glue functions have the following responsibilities:

1. They *translate* master's constraints to the corresponding constraints for each slave and vice versa; and
2. They *filter* constraints to determine which of them should be passed from the master to the slaves and which should not, and vice-versa.

In details, such glue functions are:

1. *master_to_slave(Slave,C)* that translates a master constraint C to the corresponding slave constraint, or to *true* if C is not a constraint to be sent to slave *Slave*;
2. *slave_to_master(Slave,D)* acts as the opposite of *master_to_slave* and it translates a slave constraint D to the corresponding master constraint;
3. *which_type(C)* that allows C to be classified as belonging to one of the four types of constraints mentioned above (*MASTER*, *SLAVE*, *BOTH* and *UNKNOWN*); and
4. *which_slave(C)* that maps C to a possibly empty set of slaves that can handle it.

Procedures *master_to_slave* and *slave_to_master* take also care of mapping master's variables to slaves' counterparts, and vice versa.

Few other procedures are provided by the slave interface to handle nondeterminism and will be discussed in subsection 3.3.

3.2 The Master's Constraint Solving Procedure

The procedure *step*—see Algorithm 2—is the core of the solving process of the master. It embodies all actions necessary to solve each atomic constraint in

Algorithm 2 The core of the solving procedure

```
procedure step(Constraint Store)
    Type T;
    Constraint C, D;
    Set Slaves, Selected_Slaves;

    C ← extract(Store);
    Selected_Slaves ← ∅;

    while C ≠ true do
        T ← which_type(C);
        Slaves ← which_slave(C);
        Selected_Slaves ← Selected_Slaves ∪ Slaves;

        if T = SLAVE or T = BOTH then
            for all Slave ∈ Slaves do
                Slave.add(master_to_slave(Slave, C))
            end for
        end if;

        if T = BOTH or T = MASTER then
            handle_constraint(Store, C)
        end if;

        C ← extract(Store);
    end while;

    for all Slave ∈ Selected_Slaves do
        Slave.solve(REDUCE);

        slave_try_next(Slave, Store)
    end for;
end procedure;
```

Store, either by using its own procedures or by interacting with the slave solvers. Eventually *step* modifies the constraint store *Store* as a side effect.

The invocation *extract(Store)* removes the current atomic constraint C from *Store* and returns it, moving the current atomic constraint to point to the next atomic constraint in *Store*. Then *step* has the task of deciding whether the selected constraint C must be allocated to some slave solver or not. More precisely, if the master decides—invoking procedures *which_type* and *which_slave*—to solve an atomic constraint on its own, it exploits the procedure *handle_constraint*; after its complete execution, either the atomic constraint is in an irreducible form, or it will be further processed at next step. On the contrary, if the master decides to delegate an atomic constraint to a group of slaves it posts the (translated version of the) selected constraint C to each slave in the group. A specific instance of procedure *handle_constraint* will be shown in next section, when presenting our case study.

After the whole constraint store of the master solver has been examined, the procedure *step* requests to the slaves to solve the constraints possibly posted to them. This is obtained by invoking the procedure *solve* for each slave solver in the set *Selected_Slaves*. The subsequent invocation of the procedure *slave_try_next*—see Algorithm 3—allows the master to get the results from the slave solver and to add them to its constraint store. This procedure also takes care of the possible nondeterminism occurring in the slave constraint solving.

Algorithm 3 The procedure used to explore nondeterministic choices left open by slaves

```
procedure slave_try_next(Solver Slave, Constraint Store)
    Constraint D;

    either
        D ← slave_to_master(Slave, Slave.get_constraints());

        if D = false then
            fail
        else
            insert(Store, D);
        end if;
    orelse                                          ▷ try next nondeterministic alternative
        Slave.next_solution();

        slave_try_next(Slave, Store)
    end either
end procedure;
```

3.3 Handling Nondeterminism

In our architecture we assume that both the master and the slave solvers can be sources of nondeterminism and we need to take care of both possibilities in a coherent way.

The fact that slave solvers can be nondeterministic requires that the slave interface modules further provide the following methods:

1. *next_solution()* to explore the next nondeterministic alternative that a previous call to *solve* might have left open.
2. *save()* that returns a snapshot of the current state of computation of a slave; and
3. *restore()* that restores a previously saved state.

save() and *restore()* will allow saving and restoring of the constraint store of the slave solvers whenever a choice point is detected in the rewriting process of a constraint of the master.

To be able to explore nondeterministic choices possibly left open by the slave solvers the master uses the procedure *slave_try_next*. This procedure uses the construct *either-orelse* to manage the nondeterminism that a previous call to procedure *solve* might have created. The idea of this construct (see [1]) is that the *either* branch is explored on first and just before executing it, the complete state of computation is pushed onto a backtracking stack. Subsequent branches of nondeterminism are triggered by a call to *fail* and they will execute the first unexplored *orelse* branch until no choices are left open. Before executing any *orelse* branch, the backtracking stack is used to restore the computation state, so that all branches start from the same state.

Furthermore, the procedures that handle the backtracking stack in the master solver need to be slightly modified. As a matter of fact, our use of the construct *either-orelse* is based on the assumption that the *either* branch can save the complete computation state of all slaves, and that the *orelse* branch can restore it. In details, whenever the master has to add a choice point, saving its computation state (in particular, its constraint store) onto its backtracking stack, it

also save the state of all the slave solver, as obtained by invoking the relevant *save* methods. Whenever the master solver backtracks to an unexplored choice point, it pops the computation state from the stack, including the saved states of the slaves, and it automatically forces the latter to turn back to their saved state, by invoking the relevant *restore* methods.

It is worth noting that methods *save* and *restore* are not normally explicitly considered when dealing with cooperative constraint solvers and the work is under the (implicit) assumption that solvers are implemented in a programming language that supports nondeterminism. Under this assumption, any change in the computation state of any slave occurred since last choice point is automatically undone when the master rolls back (this is the case of the Prolog implementation of CLP($\mathcal{SET} + \mathcal{FD}$) [4]). This is obviously false in a programming language like Java that do not support nondeterminism and we need to explicitly take care of it.

4 A Case Study: JSetL+JFD

In this section we describe the integration of two constraint solvers, namely JSetL and JFD, as an instance of the generic master-slave architecture described above. In such instance we assume that one single slave is used and that the implementation language, i.e., Java, does not support nondeterminism. The presented technique can be easily generalized to the case of many slaves.

JSetL is a Java library that endows Java with a number of facilities to support general purpose declarative programming like those usually found in *Constraint Logic Programming* (*CLP*) languages. In particular, JSetL provides logical variables, unification, list and set data structures, constraint solving and nondeterminism, like those supported by CLP($\mathcal{SET}$) [5].

As noted in [11], computation efficiency is not a primary design requirement of JSetL. JSetL is mainly conceived as a tool for rapid prototyping, where easiness of program development and program understanding prevail over efficiency. Moreover, JSetL is meant to provide researchers with a study tool for all issues related to set constraint solving. This is the reason why JSetL provides rich and efficient techniques for managing set variables and constraints, while it relies on basic *generate & test* constraint solving for the case of scalar variables.

The Java Finite Domains (JFD) is a library that provides the programmer with an object-oriented view of well-known constraint solving techniques for variables with finite domains. In details, it implements the CLP($\mathcal{FD}$) language for integers [2] and it provides some facilities for inspecting the computation of the solver and for integrating it with third party Java objects.

The language that JSetL provides for the definition of constraints is a superset of JFD's one and therefore JSetL can play the role of master, while JFD becomes the one and only slave. This choice requires some minor modifications to JSetL, but it saves us the trouble of realizing a new meta-solver from scratch. The result of this integration, namely JSetL+JFD, has the best of both worlds:

Algorithm 4 The *handle_constraint* procedure of JSetL+JFD

```
procedure handle_constraint(Constraint Store, Constraint C)
    if C = op∈(o1,o2) then
        member(Store, C)
    else if C = op∉(o1,o2) then
        equals(Store, C)
    else if ... then
    else if C = next then
        slave_try_next(Store, C)                    ▷ Unroll nondeterministic choice
    end if
end procedure
```

it retains the expressive power and flexibility of JSetL, while allowing an efficient finite-domains processing when needed.

The synergic cooperation of JSetL and JFD is completely transparent to the programmer because he/she deals only with the API, i.e., the language, that JSetL provides. The presence of JFD as a back-end slave is perceived only at runtime because of the improvement of performances that it brings when dealing with finite domains variables: whenever JSetL detects constraints that JFD can handle efficiently, e.g., membership constraints over finite sets of integers, it delegates their resolution to JFD.

Algorithm 4 shows (part of) the instance of procedure *handle_constraint* in the case of JSetL+JFD. *handle_constraint* implements the constraint handling for master's constraints, using a dedicated procedure for each different type of constraint. One of these procedures, namely *member*, dedicated to the management of $op_{\in}(x, y)$ constraints, is shown in Algorithm 5.

Following [4], we want our integrated solver JSetL+JFD to preserve the completeness property that characterizes CLP($\mathcal{SET}$) and JSetL. To this end, we force the slave solver JFD to label variables before returning its results to the master. To obtain this, the call *Slave.solve(REDUCE)* of Algorithm 2 is replaced by the call:

jfd.solve(LABEL_ALL)

Forcing labeling, allows the slave solver JFD to communicate back to the master solver JSetL only equality constraints, provided JFD has received enough information to associate a domain to each variable it has to deal with.

The main differences with respect to the general scheme of Section 3 is the way nondeterminism is handled within the master solver. The problem is how the abstract construct *either-orelse* can be rendered concretely using JSetL facilities for nondeterminism handling.

JSetL allows to express nondeterminism by explicitly creating choice points through the use of the method *add_choice_point*, and to backtrack to one of the open choice points using the control information stored in the constraint itself and obtainable through the method *get_alternative* of the class *Constraint*. Nondeterminism in JSetL, however, is confined to constraint solving, i.e., the aforementioned methods can be only used within constraint solving procedures (either predefined or user defined).

Algorithm 5 The *member* procedure of JSetL+JFD.

```
procedure member(Constraint Store, Constraint C)
    if C = op∈(o1, ∅) then
        fail
    else if C = op∈(o1, X) then
        insert(Store, op=(X, {o1 | N}))
    else if C = op∈(o1, {o2 | s}) then
        if C.get_alternative() = 0 then
            add_choice_point(C);                    ▷ Save nondeterministic choice

            insert(Store, op=(o1, o2))
        else
            insert(Store, op∈(o1, s))
        end if
    else if ... then
    end if
end procedure
```

As an example, consider the implementation of the constraint rewriting procedure *member* shown in Algorithm 5. Such a procedure is a source of nondeterminism because it addresses two possibilities when the constraint it handles is $C = op_{\in}(o_1, \{o_2 \mid s\})$:

$$o_1 = o_2 \vee o_1 \in s$$

The nondeterministic alternatives are implemented as the different alternatives of nested *if-else* statements, which are selected using the control information obtained through the method *get_alternative*. Each *if-else* alternative, but the last one, creates a choice point and adds it to the backtracking stack by invoking the method *add_choice_point*. Then the remaining code of the *if-else* alternative implements the proper constraint rewriting.

Algorithm 6 shows the refined versions of the procedure *slave_try_next* for JSetL+JFD. To implement the nondeterministic computation of Algorithm 3 we introduce a new constraint, called *next*, and we turn the procedure *slave_try_next* into a constraint rewriting procedure called to handle the new constraint *next*. The way this constraint is processed is exactly the same shown in procedure *slave_try_next* of Algorithm 3 except that, after calling *next_solution* of the slave, it inserts again the constraint *next* in its current constraint store *Store*. Some minor modifications of the procedure *handle_constraint* are also required to account for the new constraint *next*—see Algorithm 4. Similarly, the invocation *slave_try_next(Slave,Store)* in the procedure *step* is replaced by the invocation *insert(Store,next)* that adds the constraint *next* to the constraint store of the master.

Furthermore we assume that the procedures that handle the backtracking stack in JSetL are slightly modified to allow the JFD state to be saved/restored to/from the JSetL backtracking stack, according to the technique described in Section 3.3. This requires that JFD is equipped with the *save* and *restore* procedures described in the general case. Using this technique, the implementation of nondeterministic constraint rewriting procedures, such as member, remains completely unchanged with respect to the case of a stand-alone master.

Algorithm 6 The *slave_try_next* procedure of JSetL+JFD

```
procedure slave_try_next(Constraint Store, Constraint C)
    Constraint D;

    if C.get_alternative() = 0 then
        add_choice_point(next);                          ▷ save computation state

        D ← slave_to_master(jfd, jfd.get_constraints());

        if D = false then
            fail
        else
            insert(Store, D)
        end if
    else
        jfd.next_solution();                             ▷ try next nondeterministic alternative
        insert(Store, next)
    end if
end procedure;
```

5 Related Work

The possibility of synergically orchestrating the work of different solvers in order to efficiently overcome the limitations of each single solver is normally explored in the area of *cooperative constraint solving*.

The literature on cooperative constraint solving tries to capture this idea by designing a *composite constraint solver* that provides a unified view of the languages and the features that atomic constraint solvers offer. The composite solver has the responsibilities of all cooperation-related tasks, e.g., providing a unified constraint store, managing the communication and task breakdown between atomic solvers and translating constraints back and forth atomic solvers. Then, the language of this composite solver is likely to enrich the combination of the languages of atomic solvers with meta-level features that allow expressing strategies and heuristics for cooperation.

The literature on this subject is rich and a number of architectures have been proposed [3, 6, 9, 13]. For example, Hofstedt [9] proposes a very general, yet quite complex, architecture capable of accommodating different solvers on-the-fly in order to solve complex problems that no single solver could solve efficiently. Moreover, a substantial number of theoretical works are devoted to the study of necessary conditions for supporting efficient cooperation between different solvers [9, 10].

Many of the approaches that the literature of cooperative constraint solving proposes could be possibly applied to the case study that we consider in this work, i.e., the integration of a set solver with a finite domains solver. Nevertheless, such approaches seem to overkill the problem with very generic, yet very redundant, architectures that are not easily implementable. Moreover, many details regarding the internal mechanisms used for cooperation are not specified and it is not clear whether they may put little problems in practice, or not. Finally, most of the proposals found in the literature assume that composite solvers and atomic solvers are all implemented in a (single) logic language. This hidden

hypothesis has a strong impact when implementing solvers in an object-oriented language like Java because most of the issues related to nondeterminism must be treated explicitly.

A related research area uses the synergic cooperation of constraint-solvers to deal with *Distributed Constraint Satisfaction Problems* (see, e.g., [12]), i.e., constraint satisfaction problems that can be solved by means of an efficient distribution of tasks to solver agents. Normally, all agents have the same characteristics and a single agent would be able to efficiently solve the whole problem. The distribution is meant to boost performances and it does not deal with different capabilities agents may have. This is significantly different from the main objective of our work that targets the integration of solvers with different capabilities. We need the cooperation of solvers because no single solver is capable of efficiently addressing the whole complexity of the problem at hand; not because a single solver is not quick enough for our purposes.

The basic motivation of our work lies in the working assumption that we are not probably going to deliver *the* perfect constraint solver capable of meeting all requirements of future applications. This lead us to the study of cooperative solvers as a means to overcome the limitations of each solver. For the sake of completeness, we shall mention that the same working hypothesis led other researcher towards the realization of tools for the ad-hoc construction of application-specific solvers. For example, *Constraint Handling Rules* (*CHR*s) [7] is a language for the realization (from scratch) of the right solver for the problem at hand.

6 Conclusions and Future Work

This paper gathers ideas and results obtained from a very practical experience: the integration of JSetL and JFD into a value-added Java constraint solver that can fully exploit the expressive power of JSetL, e.g., set variables and constraints, and the efficiency of JFD in treating finite integer domains variables. The outcome of this experience is twofold:

1. A generic master-slave architecture for the integration of constraint solvers with different capabilities has been identified and studied; and
2. The JSetL+JFD Java library for constraint solving is implemented.

Future work is in the direction of improving the generic architecture that we defined by relaxing some design choices that may prohibit the use of it in many interesting situations. For example, we are in the process of changing the policy for allocation of constraints to slaves in order to make it more flexible and dynamic.

From the point of view of the integration of set and finite domains constraints, we envisage the possibility of allowing the manipulation of finite domains by means of JSetL high-level set operations. The importance of this feature has been emphasized by various authors (see, e.g., [8]). In practice this would be achieved by implementing the constraint rewriting procedures defined in [4].

Other interesting directions of future development are inherent to the choice of forcing slaves, when possible, to label their variables before returning their results to the master, in order to gain completeness. This labeling, however, would probably cause inefficiencies. A first cut to such inefficiencies could be achieved by carefully deciding when to ask slaves to label variables and which variables they should label. Obviously, the more we delay the labeling request, the better. Then, a possibly larger improvement of efficiency could be achieved by dropping the completeness requirement to let slaves deciding the best way to process their constraint stores.

Acknowledgments The work is partially supported by MIUR project "Constraints and preferences as a unifying formalism for system analysis and solution of real-life problems".

References

1. K. R. Apt, J. Brunekreef, V. Partington and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5), 1014–1066, 1998.
2. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3), 185–226, 1996.
3. M. Correia, P. Barahona and F. Azevedo. CaSPER: A programming environment for development and integration of constraint solvers. *Procs.* 1^{st} *International Workshop on Constraint Programming Beyond Finite Integer Domains* (*BeyondFD'05*), 59–73, 2005.
4. A. Dal Palù, A. Dovier, E. Pontelli and G. Rossi. Integrating finite domain constraints and CLP with sets. *Procs.* 5^{th} *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (*PPDP'03*), ACM Press, 219–229, 2003.
5. A. Dovier, C. Piazza, E. Pontelli and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.
6. S. Frank, P. Hofstedt and P. R. Mai. A flexible meta-solver framework for constraint solver collaboration. *Procs.* 26^{th} *German Conference on Artificial Intelligence*, *KI'2003*, LNCS 2821, Springer-Verlag, 2003.
7. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3), 1998.
8. M. Gavanelli, E. Lamma, P. Mello and M. Milano. Dealing with incomplete knowledge on CLP(FD) variable domains, In *ACM TOPLAS*, 27(2):236–263, 2005.
9. P. Hofstedt. Cooperating constraint solvers. *Procs. International Conference on Principle and Practice of Constraint Programming*, LNCS 1894, Springer-Verlag, 520–524, 2000.
10. E. Monfroy and C. Castro. Basic components for constraint solver cooperations. *Procs. ACM SAC 2003*, ACM Press, 367–374, 2003.
11. G. Rossi, E. Panegai and E. Poleo. JSetL: A Java library for supporting declarative programming in Java. *Software-Practice & Experience*, in print, 2006.
12. A. Petcu, B. Faltings and D. Parkes. MDPOP: Faithful distributed implementation of efficient social choice problems. *Procs. Autonomous Agents and Multiagent Systems* (*AAMAS'06*), 2006.
13. P. Zoeteweij and F. Arbab. A component-based parallel constraint solver. *Procs. of COORDINATION 2004*, LNCS 2949, 307–322, Springer-Verlag, 2004.