# A Framework for Cooperating Solvers – A prototypic Implementation

Petra Hofstedt, Dirk Seifert, Eicke Godehardt
{ph,dseifert,icke}@cs.tu-berlin.de

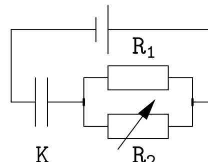
Berlin University of Technology

**Abstract.** This paper describes a prototypic implementation of our framework of cooperating constraint solvers. The system allows the integration of arbitrary constraint solvers providing typical interface functions. This enables to build constraint solver cooperations according to current requirements and, thus, comfortable modelling and solving of a wide range of problems.

## 1 Introduction

The paradigm of constraint programming offers efficient mechanisms to handle constraints of various constraint domains. However, it has been shown to be desirable to combine several constraint solving techniques because this combination makes it possible to solve problems that none of the single solvers can handle alone.

*Example 1.* Let an electric circuit be given with a resistor $\mathtt{R}_1$ of $0.1M\Omega$ connected in parallel with a variable resistor $\mathtt{R}_2$ of between $0.1M\Omega$ and $0.4M\Omega$, a capacitor $K$ is in series connection with the two resistors.




Also, there is a kit of electrical components in which capacitors of $1\mu F, 2.5\mu F, 5\mu F, 10\mu F, 20\mu F$, and $50\mu F$ are available. We want to know which capacitor to use in our circuit such that the time until the voltage of the capacitor reaches 99% of the final voltage is between 0.5s and 1s, i.e. the duration until the capacitor is loaded is between 0.5s and 1s.

Thus, the input constraint conjunction is $\mathtt{R}_1 = 10^5 \wedge \mathtt{R}_2 = [10^5, 4 \cdot 10^5] \wedge (1/\mathtt{R}) = (1/\mathtt{R}_1) + (1/\mathtt{R}_2) \wedge \mathtt{V}_\mathtt{K} = \mathtt{V} \times (1 - exp(-\mathtt{t}/(\mathtt{R} \times \mathtt{K}))) \wedge \mathtt{V}_\mathtt{K} = 0.99 \times \mathtt{V} \wedge \mathtt{t} = [0.5, 1] \wedge \mathtt{K} \in \{10^{-6}, 2.5 \cdot 10^{-6}, 5 \cdot 10^{-6}, 10^{-5}, 2 \cdot 10^{-5}, 5 \cdot 10^{-5}\}$.

This constraint conjunction can be solved using different cooperating constraint solvers. A solver for rational interval arithmetic infers from the constraint conjunction $\mathtt{R}_1 = 10^5 \wedge \mathtt{R}_2 = [10^5, 4 \cdot 10^5] \wedge (1/\mathtt{R}) = (1/\mathtt{R}_1) + (1/\mathtt{R}_2)$ the new constraint $\mathtt{R} = [5 \cdot 10^4, 8 \cdot 10^4]$. Let us assume that, since the interval solver is able to handle constraints of rational arithmetic, it also computes from $\mathtt{V}_\mathtt{K} = \mathtt{V} \times (1 - \mathtt{E}) \wedge \mathtt{V}_\mathtt{K} = 0.99 \times \mathtt{V}$, where $\mathtt{E} = exp(-\mathtt{t}/(\mathtt{R} \times \mathtt{K}))$, the new constraint $\mathtt{E} = 0.01$. This constraint is given to a constraint solver which is able to handle functions, like *exp*, *ln*, *sin*, and *cos*, to infer from the constraint conjunction $\mathtt{E} = 0.01 \wedge \mathtt{E} = exp(\mathtt{F})$, where $\mathtt{F} = -\mathtt{t}/(\mathtt{R} \times \mathtt{K})$, the constraint $\mathtt{F} = ln(0.01)$. This constraint cannot be understood by the interval solver, however, we are able to infer from $\mathtt{F} = ln(0.01)$

the (slightly weaker) interval constraint $\texttt{F} = [-4.6052, -4.6051]$. Now, the interval solver is able to compute the constraint $1.3571 \cdot 10^{-6} \leq \texttt{K} \leq 4.3431 \cdot 10^{-6}$ from the constraints $\texttt{F} = [-4.6052, -4.6051]$, $\texttt{F} = -\texttt{t}/(\texttt{R} \times \texttt{K})$, $\texttt{R} = [5 \cdot 10^4, 8 \cdot 10^4]$, and $\texttt{t} = [0.5, 1]$. The last step is done by a finite domain constraint solver which uses the constraints $\texttt{K} \in \{10^{-6}, 2.5 \cdot 10^{-6}, 5 \cdot 10^{-6}, 10^{-5}, 2 \cdot 10^{-5}, 5 \cdot 10^{-5}\}$ and $1.3571 \cdot 10^{-6} \leq \texttt{K} \leq 4.3431 \cdot 10^{-6}$ to choose the capacitor which we are searching for from the kit: $\texttt{K} = 2.5 \cdot 10^{-6}$ holds. □

In [Hof00,Hof01] we introduced a flexible combination mechanism for constraint solvers of different constraint systems. In this paper, we consider a prototypic implementation of our overall framework for cooperating constraint solvers.

This work is organized as follows: After a short introduction into basic concepts of constraint programming in Sect.2. we shortly recall our system for cooperating constraint solvers in Sect.3. At this, we only touch the definition of the behaviour of the overall system by means of a reduction relation as far as this is necessary to understand our contributions. The introduced uniform interface for constraint solvers is the basis of the information exchange between the cooperating solvers. Section 4 is dedicated to the prototypic implementation of our framework of cooperating constraint solvers. We introduce the solvers which have been chosen for cooperation in Sect.4.1. The realization of our cooperation approach is treated in Sect.4.2. At this, we go into detail w.r.t. the implemented control mechanism. In Sect.4.3, we demonstrate by means of examples the possibilities of system configuration. We discuss the implementation w.r.t. the theoretical approach in Sect.5.

## 2 Constraint Programming and Constraint Solvers

A *signature* $\Sigma = (S, F, R; ar)$ consists of a set $S$ of sorts, a set $F$ of function symbols, a set $R$ of predicate symbols, and an arity function $ar: F \cup R \to S^\star$. $S$, $F$ and $R$ are mutually disjoint. A set of variables appropriate to $\Sigma$ is a many sorted set $X = \bigcup_{s \in S} X^s$, where $\forall s \in S$ the set $X^s$ is countably infinite. A *$\Sigma$-structure* $\mathcal{D} = (\{\mathcal{D}^s \mid s \in S\}, \{f^\mathcal{D} \mid f \in F\}, \{r^\mathcal{D} \mid r \in R\})$ consists of an $S$-sorted family of nonempty carrier sets $\mathcal{D}^s$, a family of functions $f^\mathcal{D}$, and a family of predicates $r^\mathcal{D}$ appropriate to $F$ and $R$. Let the set of terms $\mathcal{T}(F, X)$ be defined as usually. In the following, we assume familiarity with the fundamentals of predicate logic.

**Definition 1 (constraint, constraint system).** *Let $\Sigma = (S, F, R; ar)$ be a signature, where $R$ contains at least one predicate symbol $=^s_\Sigma$ for every $s \in S$. Let $X$ be a set of $\Sigma$-variables. Let $\mathcal{D}$ be a $\Sigma$-structure with equality, i.e. for every predicate symbol $=^s_\Sigma$ there is a predicate $=^s_\mathcal{D} \subseteq \mathcal{D}^s \times \mathcal{D}^s$, for which the usual axioms for equality hold.*

*A* constraint *is a string $r(t_1, \ldots, t_m)$, where $r \in R$ with $ar(r) = s_1 \ldots s_m$ and $t_i \in \mathcal{T}(F, X)^{s_i}$. The set of constraints over $\Sigma$ is denoted by $\mathcal{C}onstraint$. It contains, furthermore, the two distinct constraints $true$ and $false$ with $\mathcal{D} \vDash true$ and $\mathcal{D} \nvDash false$. The 4-tupel $\zeta = (\Sigma, \mathcal{D}, X, \mathcal{C}ons)$, where $\{true, false\} \subseteq \mathcal{C}ons \subseteq \mathcal{C}onstraint$, is a* constraint system. □

A *solution* of a disjunction of constraint conjunctions $C$ in $\mathcal{D}$ is a valuation $\sigma : Y \to \mathcal{D}$ where $var(C) \subseteq Y \subseteq X$, such that $(\mathcal{D}, \sigma) \vDash C$ holds. *Solving* the

disjunction $C$ means finding out whether there is a solution for $C$ or not, i.e. finding out whether $C$ is satisfiable in $\mathcal{D}$ or not.

Given a constraint system, we need appropriate algorithms for constraint manipulation. A *constraint solver* $CS$ is associated with a constraint system $\zeta$. It is a collection of operations on *disjunctive constraints,* i.e disjunctions of constraint conjunctions, of the associated constraint system. Typically a constraint solver consists of a combination of instantiations of the operations constraint satisfaction, constraint entailment, projection and simplification.

## 3 Cooperating Constraint Solvers

We let constraint solvers cooperate to make it possible to solve problems which none of the single solvers can handle alone. Figure 1 shows the architecture of our overall system for cooperating solvers. Let $L$ be the set of indices of constraint systems, $\mu, \nu \in L$. To every individual solver $CS_\nu$ a *constraint store* $C^\nu$ is assigned. Let $\mathcal{DCC}ons_\nu$ denote the set of disjunctive constraints of $\zeta_\nu$. A constraint store $C^\nu \in \mathcal{DCS}tore_\nu \subseteq \mathcal{DCC}ons_\nu$ is a disjunctive constraint which is satisfiable in the corresponding structure. The *meta constraint solver* coordinates the work of the different individual solvers and it manages the *constraint pool.* Initially, the constraint pool contains the constraints of the constraint conjunction $\Phi$ which we want to solve. The meta solver takes constraints from the constraint pool and

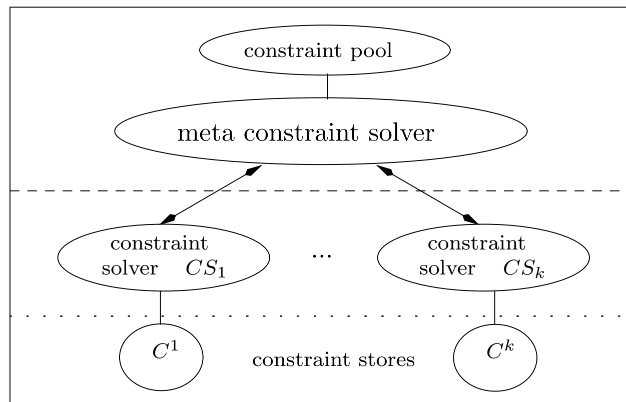



**Fig. 1.** Architecture of the overall system

passes them to the constraint solvers of the corresponding constraint domains (step 1). The individual solvers propagate the received constraints to their stores (step 2). The meta solver forces them to extract information from their constraint stores. This information is added by the meta solver to the constraint pool (step 3). The procedure of steps 1-3 is repeated until the pool contains either the constraint $false$ or the constraint $true$ only. If the constraint pool contains $false$ only, then the initially given conjunction $\Phi$ of constraints is unsatisfiable. If the pool contains $true$ only, then the system could not find a contradiction. Solutions of $\Phi$ can

be retrieved from the current constraint stores. Using the described mechanism, because of information exchange between the solvers, each individual solver deals with more information than only that of its associated constraints of $\Phi$.

*Syntax* Since we want to solve mixed disjunctive constraints such that every constraint may contain function symbols and predicate symbols of different constraint systems it is necessary to convert every such disjunction into a disjunction such that every constraint is defined by function symbols and predicate symbols of exactly one constraint system. This is done by flattening. In [Hof01] we give a definition of the function *Flatten* and we show that the set of solutions w.r.t. the common variables is preserved. Thus, after flattening we can solve the newly built disjunction instead of the original mixed one.

*Example 2.* Given two constraint systems $\zeta_\nu = (\Sigma_\nu, \mathcal{D}_\nu, X_\nu, \mathcal{C}ons_\nu)$ with $\Sigma_\nu = (S_\nu, F_\nu, R_\nu; ar_\nu)$, $\nu \in \{1,2\}$, where $=_\nu \in R_\nu$, $\{-, /, \times\} \subseteq F_1 \backslash F_2$ and $\texttt{exp} \in F_2 \backslash F_1$ hold. The constraint $(\texttt{E} =_2 \texttt{exp}(\texttt{0} - \texttt{t}/(\texttt{R} \times \texttt{K})))$ is a mixed constraint which none of the solvers $CS_1$ and $CS_2$ is able to handle. We may perform the following transformation:
$Flatten(\texttt{E} =_2 \texttt{exp}(\texttt{0} - \texttt{t}/(\texttt{R} \times \texttt{K}))) = (\texttt{E} =_2 \texttt{exp}(\texttt{X}_1)) \wedge (\texttt{X}_1 =_1 (\texttt{0} - \texttt{t}/(\texttt{R} \times \texttt{K})))$.
We got constraints which can be uniquely assigned to one constraint system $\zeta_1$ resp. $\zeta_2$: $(\texttt{X}_1 =_1 (\texttt{0} - \texttt{t}/(\texttt{R} \times \texttt{K}))) \in \mathcal{C}onstraint_1$ and $(\texttt{E} =_2 \texttt{exp}(\texttt{X}_1)) \in \mathcal{C}onstraint_2$ hold. □

### 3.1 A Uniform Interface for Constraint Solvers

To enable a cooperation, the solvers need to exchange information. Let to every constraint system a constraint solver be assigned. Consider a constraint solver $CS_\nu$, $\nu \in L$. Our *uniform interface* of $CS_\nu$ consists of a function $tell_\nu$ for *constraint propagation* (according to step 2 of the above behaviour description of the system) and a set of functions $proj_{\nu \to \mu}$ for *constraint projection* (corresponding to the above step 3).

*Constraint Propagation* The (partial) function $tell_\nu$ is due to constraint satisfaction. $tell_\nu$ adds a constraint $c \in \mathcal{C}ons_\nu$ to a constraint store $C \in \mathcal{DCS}tore_\nu$ if the conjunction of $c$ and $C$ is satisfiable, i.e. if $\mathcal{D} \vDash \exists(C \wedge c)$ holds. Figure 2 shows our requirements to the function $tell_\nu$.

(1) The first case describes the situation, where a redundant constraint $c$ is added to the constraint store $C$. At this, $C$ does not change.
(2) A constraint $c$ is added to the store. Notice, that $c$ may even be redundant w.r.t. $C$. The new store $C'$ comes from $C$ by adding knowledge from $c$. The disjunction $C''$ describes remaining constraints of $c$ to be propagated later.
(3) In this situation the conjunction of $c$ and $C$ is unsatisfiable.

Giving requirements to the interface function $tell_\nu$ instead of a definition enables the integration of a high number of existing solvers into our overall system. The requirements allow to take particular properties of solvers, like their incompleteness or an existing entailment test, into consideration for cost reduction for our overall system. For a detailed description see [Hof01].

$tell_\nu$: $\mathcal{C}ons_\nu \times \mathcal{DCS}tore_\nu \longrightarrow$
$\{true_{changed}, true_{redundant}, false\} \times \mathcal{DCS}tore_\nu \times \mathcal{DCC}ons_\nu$ with

(1) if $tell_\nu(c, C) = (true_{redundant}, C', C'')$,
then $C' = C$, $C'' = true$, and $\mathcal{D}_\nu \vDash \forall(C \longrightarrow c)$,

(2) if $tell_\nu(c, C) = (true_{changed}, C', C'')$, then
(a) $\mathcal{D}_\nu \vDash \forall((C \wedge c) \longleftrightarrow (C' \wedge C''))$, (b) $\mathcal{D}_\nu \vDash \forall(c \longrightarrow C'')$,
(c) $\mathcal{D}_\nu \vDash \forall(C' \longrightarrow C)$ and (d) $\mathcal{D}_\nu \vDash \exists C'$,

(3) if $tell_\nu(c, C) = (false, C', C'')$, then $C' = C$, $C'' = false$ and $\mathcal{D}_\nu \nvDash \exists(C \wedge c)$.

**Fig. 2.** Interface function $tell_\nu$ (requirements)

*Example 3.* Let the interface function $tell_{\mathcal{R}_{lin}}$ of a solver $CS_{\mathcal{R}_{lin}}$ of a constraint system $\zeta_{\mathcal{R}_{lin}}$ of linear constraints over real numbers be defined as follows (an according implementation is possible using the simplex algorithm):

$$\begin{aligned}
tell_{\mathcal{R}_{lin}}(c, C) &= (true_{redundant}, C, true), && \text{if } \mathcal{D} \vDash \forall(C \longrightarrow c),\\
tell_{\mathcal{R}_{lin}}(c, C) &= (true_{changed}, C', true), && \text{if } \mathcal{D} \vDash \forall((C \wedge c) \longleftrightarrow C'),\\
& && \mathcal{D} \nvDash \forall(C \longrightarrow c) \text{ and } \mathcal{D} \vDash \exists(C \wedge c),\\
tell_{\mathcal{R}_{lin}}(c, C) &= (false, C, false), && \text{if } \mathcal{D} \nvDash \exists(C \wedge c).
\end{aligned}$$

The following examples show the application of $tell_{\mathcal{R}_{lin}}$:
$tell_{\mathcal{R}_{lin}}(c_1, C) = (true_{changed}, C', true)$, where
$c_1 = (x \leq 3)$, $C = true$, $\mathcal{D}_{\mathcal{R}_{lin}} \vDash \forall(C' \longleftrightarrow (c_1 \wedge C))$.
$tell_{\mathcal{R}_{lin}}(c_2, C') = (true_{redundant}, C', true)$, where $c_2 = (x \leq 4)$ holds.
$tell_{\mathcal{R}_{lin}}(c_3, C') = (false, C', false)$, where $c_3 = (x = 4)$ holds. □

*Projection of Constraint Stores* Constraint projection is used to enable information exchange between the solvers. The function $proj_{\nu \to \mu}$ (see Fig.3) projects a constraint store $C^\nu$ w.r.t. another constraint system $\zeta_\mu$, $\mu \in L \backslash \{\nu\}$. It provides knowledge which is implied by the store $C^\nu$ of $CS_\nu$ in the form of constraints of $\zeta_\mu$. The projection function $proj_{\nu \to \mu}$ must be defined in such a way that every solution of $C^\nu$ in $\mathcal{D}_\nu$ is a solution of the projection $proj_{\nu \to \mu}(Y, C^\nu)$ in $\mathcal{D}_\mu$, where $Y \subseteq X_\nu \cap X_\mu$. This ensures that projecting a constraint store w.r.t. another constraint system, no solutions of the constraints of the store are lost. We call this required property *soundness*, its formal description can be found in [Hof01].

$proj_{\nu \to \mu}$: $\mathcal{P}(X_{\nu,\mu}) \times \mathcal{DCS}tore_\nu \to \mathcal{DCC}ons_\mu$ with
$X_{\nu,\mu} = X_\nu \cap X_\mu$, $var(proj_{\nu \to \mu}(Y, C^\nu)) \subseteq Y$.

**Fig. 3.** Interface function $proj_{\nu \to \mu}$ (requirements)

Usually $proj_{\nu \to \mu}$ will be defined by means of a projection function $proj_\nu$ projecting a store $C^\nu$ and yielding constraints of $\mathcal{DCC}ons_\nu$ and a conversion function $conv_{\nu \to \mu}$: $\mathcal{DCC}ons_\nu \to \mathcal{DCC}ons_\mu$:

$proj_{\nu\to\mu}(Y, C^\nu) = conv_{\nu\to\mu}(proj_\nu(Y, C^\nu))$ with $Y \subseteq X_\nu \cap X_\mu$.

Thus, each single constraint solver can be regarded as black box solver equipped with a projection function which allows the projection of the constraint store w.r.t. a set of variables. These black box solvers are extended by functions for converting projections w.r.t. other constraint systems. The aim of projecting a disjunctive constraint $C \in \mathcal{DC}Cons_\nu$ w.r.t. a sequence $\tilde{Y}$ (with $Y \subseteq X$) of variables which occur in $C$ is to find a disjunctive constraint $C'$ which is equivalent to $\exists_{_\tilde{Y}} C$ and where the variables which do occur in $C$ but not in $\tilde{Y}$ are eliminated: $\mathcal{D}_\nu \vDash \forall(\exists_{_\tilde{Y}} C \longleftrightarrow C')$. However, since sometimes it is not possible to compute $C'$ or it is not possible to compute it efficiently, we require *proj* to be defined as given in Fig.4.

$proj_\nu$: $\mathcal{P}(X) \times \mathcal{DC}Store_\nu \to \mathcal{DC}Cons_\nu$ with
$proj_\nu(Y, C) = C'$, where $\mathcal{D}_\nu \vDash \forall((\exists_{_\tilde{Y}} C) \longrightarrow C')$ and $var(C') \subseteq Y$.

**Fig. 4.** Interface function $proj_\nu$ (requirements)

*Example 4.* Consider the solver $CS_{\mathcal{R}_{lin}}$ and a solver $CS_{FD}$ of a finite domain constraint system $\zeta_{FD}$. Let $C^{FD} = ((y =_{FD} 3) \wedge (x >_{FD} y) \wedge (x \in_{FD} \{2,3,4,5,6\}))$ hold. The projection functions $proj_{FD}$ and $proj_{FD\to\mathcal{R}_{lin}}$ of $CS_{FD}$ could work as follows:

$$proj_{FD}(\{x\}, C^{FD}) = (x \in_{FD} \{4,5,6\}) \text{ and}$$
$$\begin{aligned} proj_{FD\to\mathcal{R}_{lin}}(\{x\}, C^{FD}) &= conv_{FD\to\mathcal{R}_{lin}}(proj_{FD}(\{x\}, C^{FD})) \\ &= ((x \geq 4) \wedge (x \leq 6)). \end{aligned}$$ □

In the following, we require given computable functions $tell_\nu$ and $proj_{\nu\to\mu}$, $\nu, \mu \in L$.

### 3.2 Description of the System Behaviour

The behaviour of our system is described by means of reduction relations for *overall configurations*. An overall configuration $\mathcal{H}$ consists of a *formal disjunction* $\dot{\bigvee}_{i\in\{1,\ldots,m\}} \mathcal{G}_i$ of configurations $\mathcal{G}_i$. Formal disjunction $\dot{\vee}$ is commutative and associative. A *configuration* $\mathcal{G} = (\mathcal{P} \odot \bigwedge_{\nu\in L} C^\nu)$ corresponds to the architecture of the overall system (Fig.1). It consists of the constraint pool $\mathcal{P}$ which is a set of constraints which we want to solve and the conjunction $\bigwedge_{\nu\in L} C^\nu$ of constraint stores. In [Hof00] we show elaborately how to define *strategies for cooperating constraint solvers,* i.e. reduction systems for overall configurations using the interface functions of the solvers. In the following, we shortly show the general procedure.

In general, in one derivation step one or more configurations $\mathcal{G}_i$, $i \in \{1,\ldots,m\}$, are rewritten by a formal disjunction $\mathcal{HG}_i$ of configurations:

$$\begin{aligned} OConf1 &= \mathcal{H}_1 \dot{\vee} \mathcal{G}_1 \;\dot{\vee} \ldots \dot{\vee} \mathcal{H}_i \dot{\vee} \mathcal{G}_i \;\dot{\vee} \ldots \dot{\vee} \mathcal{H}_m \dot{\vee} \mathcal{G}_m \;\dot{\vee} \mathcal{H}_{m+1} \implies \\ OConf2 &= \mathcal{H}_1 \dot{\vee} \mathcal{HG}_1 \dot{\vee} \ldots \dot{\vee} \mathcal{H}_i \dot{\vee} \mathcal{HG}_i \dot{\vee} \ldots \dot{\vee} \mathcal{H}_m \dot{\vee} \mathcal{HG}_m \dot{\vee} \mathcal{H}_{m+1} \end{aligned}$$

Thus, first, we define a derivation relation for configurations and, based on this, we define a derivation relation for overall configurations.

*Step 1. Definition of a derivation relation for configurations (production level).* The simplest possibility to define a derivation step $\mathcal{G}_i \to \mathcal{HG}_i$ is to take exactly one constraint $c \in \mathcal{C}ons_\nu$, $\nu \in L$, from the constraint pool of $\mathcal{G}_i$ and to propagate it to its associated store $C^\nu$ (using $tell_\nu$). This is followed by projections of the newly built store $C'^\nu$ w.r.t. other constraint systems building the new overall configuration $\mathcal{HG}_i$ (using $proj_{\nu\to\mu}$). It is possible to define many other strategies for the production level in this way, like for example parallel work or fixed orders for the solvers to work or for constraints to be propagated next.

*Step 2. Defining a derivation relation for overall configurations (application level).* A derivation step $OConf1 \Longrightarrow OConf2$ for overall configurations is defined on the basis of the derivation relation for configurations (at production level). There are as well many possibilities: for example, we may define a derivation step such that the derivation of exactly one configuration or the parallel derivation of several configurations is allowed.

Using this *two-step frame* different reduction systems which realize different cooperation strategies for the solvers have been described (see [Hof00]). The reduction systems allow the derivation of an *initial overall configuration* $\mathcal{G}_0 = \mathcal{P}_\Phi \odot \bigwedge_{\nu\in L} C_0^\nu$, where the constraint pool $\mathcal{P}_\Phi$ contains the constraints of the conjunction $\Phi$ which we want to solve and all constraint stores $C_0^\nu$, $\nu \in L$, contain the constraint *true* only. From the derived normal form we obtain information about the satisfiability of the initially given disjunctive constraint.

In [Hof01] we formally proved termination and confluence, soundness and completeness of the defined reduction systems which describe the behaviour of the system. We analysed the computation results using our overall system of cooperating solvers and showed that the information exchange between the solvers enables our overall system to solve disjunctions of constraint conjunctions which the single solvers are not able to handle. We used our combination mechanism for the definition of different cooperation strategies (see also [Hof00]) and we showed the transfer of the theoretical results w.r.t. solutions, termination, and confluency to the newly defined strategies.

# 4 Implementation

In this section we introduce a prototypic implementation of our framework of cooperating constraint solvers. We introduce the solvers which have been chosen for cooperation in Sect.4.1, describe the realization of our cooperation approach in Sect.4.2, and demonstrate the possibilities of system configuration by means of examples in Sect.4.3.

## 4.1 Integrated Solvers

Typical applications of cooperating solvers are arithmetic problems. Thus, the constraint systems and the associated solvers to be integrated have been chosen accordingly. Three freely available constraints solvers have been chosen for cooperation: a constraint solver for linear constraints over rational numbers, a finite domain constraint solver, and an interval constraint solver.

*The Solver for Linear Arithmetic* The constraint solver LinAr for linear constraints over rational numbers has been developed by Olaf Krzikalla as a study work in [Krzan]. LinAr is based on the simplex algorithm and it has been implemented in the language C. This solver is able to handle linear equations, inequalities, and disequations over rational numbers. It handles nonlinear constraints by a delay mechanism. The constraint system of LinAr is defined as follows:

$\zeta_{LinAr} = (\Sigma_{LinAr}, \mathcal{D}_{LinAr}, X_{LinAr}, \mathcal{C}ons_{LinAr})$, where

$\Sigma_{LinAr}$ consists of the constants (i.e. 0-ary function symbols) 0, 1, 2, 3, ..., the binary function symbols $+$, $-$, and $\times$ and the binary predicate symbols $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$:

$$\begin{aligned}\Sigma_{LinAr} &= (S_{LinAr}, F_{LinAr}, R_{LinAr}; ar_{LinAr}) \\ &= (\{rat\}, \{0, 1, 2, 3, \dots, +, -, \times\}, \{=, \neq, <, \leq, >, \geq\}; ar_{LinAr}).\end{aligned}$$

$\mathcal{D}_{LinAr}$ is defined as expected.

The interface function $tell_{LinAr}$ is defined according to $tell_{\mathcal{R}_{lin}}$ of Example 3. The function $proj_{LinAr}$ projects $r$-relations, with $r \in \{=, <, \leq, >, \geq\}$, between variables and terms.

*The Finite Domain Constraint Solver* On the base of a free library [CSP01] of routines for solving binary constraint satisfaction problems from Peter van Beek our finite domain constraint solver CSPlib has been implemented. While the routines library has been implemented in C, our solver CSPlib has been implemented in Java. The constraint system of CSPlib is the following:

$\zeta_{CSPlib} = (\Sigma_{CSPlib}, \mathcal{D}_{CSPlib}, X_{CSPlib}, \mathcal{C}ons_{CSPlib})$, where

$\Sigma_{CSPlib}$ consists of a set *Constants* of constants (i.e. 0-ary function symbols) and the binary predicate symbols $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, and *in*.

$$\begin{aligned}\Sigma_{CSPlib} &= (S_{CSPlib}, F_{CSPlib}, R_{CSPlib}; ar_{CSPlib}) \\ &= (\{const, \mathcal{P}const\}, Constants, \{=, \neq, <, \leq, >, \geq, in\}; ar_{CSPlib}),\end{aligned}$$

where $ar_{CSPlib}(=) = ar_{CSPlib}(\neq) = \ldots = ar_{CSPlib}(\geq) = const\ const$,
$ar_{CSPlib}(in) = const\ \mathcal{P}const$.

The set *Constants* of constants is not preliminarily fixed. Giving a conjunction $C$ of constraints to be solved, for every variable $x \in var(C)$ a constraint of the form $x\ in\ \{\ldots\}$ must be given which restricts the set of values for $x$ to a finite set. From these constraints the set *Constants* of constants is derived. For example, if the constraints contain numbers, then *Constants* contains the rational numbers (with the usual order) and all possible sets of rational numbers, i.e. it contains $\mathbb{Q}$ and $\mathcal{P}(\mathbb{Q})$. Thus, a constant $v \in Constants$ is either a simple value, like a number or a string, $ar_{CSPlib}(v) = const$ holds in this case, or it is a set of simple values, $ar_{CSPlib}(v) = \mathcal{P}const$ holds. $\mathcal{D}_{CSPlib}$ is defined accordingly. $\mathcal{C}ons_{CSPlib} = \mathcal{C}onstraint_{CSPlib}$ holds.

The interface function $tell_{CSPlib}$ of the constraint solver CSPlib has been defined in such a way that the propagated constraint is added to the constraint store and afterwards node-consistency checking and arc-consistency checking is performed. Checking satisfiability is delayed until a projection occurs. Obviously,

CSPLIB is incomplete. Because of the delay of satisfiability checking, the constraint store of CSPLIB may even become unsatisfiable. At first sight, this seems to be a hard variation to the *tell*-definition in Sect.3. However, since in our framework every (successful) propagation enforces a projection later on, the inconsistency is detected at the latest at the time of projecting the store. Thus, this variation produces no real problems or inconsistencies using the system w.r.t. the theoretical results or the desired behaviour of the system.

The projection function $proj_{CSPlib}$ projects *in*-relations between variables and sets of values. On top of $proj_{CSPlib}$ two further projection functions have been implemented: The projection function $weakProj_{CSPlib}$ converts the original *in*-relations between a variable $x$ and a set of values into a conjunction $a \leq x \wedge x \leq b$, where $a$ is the smallest value in the set (w.r.t. the corresponding order) and $b$ is the biggest value in the set. Here, we talk about a 'weak' projection because in the interval expressed by the resulting constraint conjunctions there may appear values which are not contained in the original set. Weak projection creates a conjunction of constraints. The projection function $strongProj_{CSPlib}$ converts the constraint $x\ in\ \{v_1, \ldots, v_n\}$ into a disjunction $x = v_1 \vee \ldots \vee x = v_n$ of equality constraints. Strong projection creates a disjunction of constraints, where, in contrast to weak projection, the projected constraints express exactly the set of values of the original *in*-constraint.

*The Interval Constraint Solver* The third solver chosen for integration into the system of cooperating constraint solvers is a solver IA which is due to a solver [Bra01] for interval arithmetic from Timothy J. Hickey from Brandeis University. The solver is implemented in JAVA. The constraint system of IA is the following: $\zeta_{IA} = (\Sigma_{IA}, \mathcal{D}_{IA}, X_{IA}, \mathcal{C}ons_{IA})$, where

$$\begin{aligned}\Sigma_{IA} &= (S_{IA}, F_{IA}, R_{IA}; ar_{IA}) \\ &= (\{real\}, \{0, 1, 2, 3, \ldots, +, -, \times, /, \hat{}, exp, log, sin, cos, \ldots\}, \\ &\quad \{=, \neq, <, \leq, >, \geq, in[_, _], \ldots\}; ar_{IA}). \\ \mathcal{D}_{IA} &= (\{\mathbb{R}\}, \{0^{\mathbb{R}}, 1^{\mathbb{R}}, 2^{\mathbb{R}}, 3^{\mathbb{R}}, \ldots, +^{\mathbb{R}}, -^{\mathbb{R}}, \times^{\mathbb{R}}, /^{\mathbb{R}}, \hat{}^{\mathbb{R}}, exp^{\mathbb{R}}, log^{\mathbb{R}}, sin^{\mathbb{R}}, cos^{\mathbb{R}}, \ldots\}, \\ &\quad \{=^{\mathbb{R}}, <^{\mathbb{R}}, \leq^{\mathbb{R}}, >^{\mathbb{R}}, \geq^{\mathbb{R}}, in^{\mathbb{R}}\}), \text{ where}\end{aligned}$$

the functions and relations are defined as expected.
For $in^{\mathbb{R}}$ holds: $in^{\mathbb{R}} \subseteq \mathbb{R} \times \mathbb{R} \times \mathbb{R}$ with $\forall x, y, z \in \mathbb{R}$: $x \in in^{\mathbb{R}}(y, z)$ iff $y \leq x \leq z$.
$\mathcal{C}ons_{IA} = \mathcal{C}onstraint_{IA}$ holds.

The solver IA is an incomplete solver. The implementation of the interface functions on top of existing functions of the solvers was simple. The implemented interface matches the theoretical requirements of Sect.3.1. The projection function $proj_{IA}$ projects constraint conjunctions of the form $a \leq x \wedge x \leq b$, where $x$ is a variable and $a$ and $b$ are values. Such a projected constraint conjunction describes an interval of admissible values of the corresponding variable.

### 4.2 Cooperation

The overall system of cooperating solvers consists, according to the overall architecture given in Fig.1, of a constraint pool, a meta constraint solver, i.e. the control

mechanism, and the individual constraint solvers. In contrast to the theoretical description which allows the handling of disjunctive constraints the implemented system may only handle an initially given constraint conjunction. If during the computation a disjunction of constraints is projected, which causes in theory the splitting of a configuration into an overall configuration, this is realized via creating choice points, saving the current computation state, and a following depth first search.

The meta constraint solver is the mechanism which realizes the cooperation strategy. This cooperation mechanism has been implemented in Java. In our implementation, because of time limitations, it has been decided in favour of one relatively fixed cooperation strategy with the possibility of some variation for experiments. An extension of the system with the possibility of a more flexible handling of strategies is a subject of current work. Figure 5 shows a flowchart of the implemented cooperation strategy.

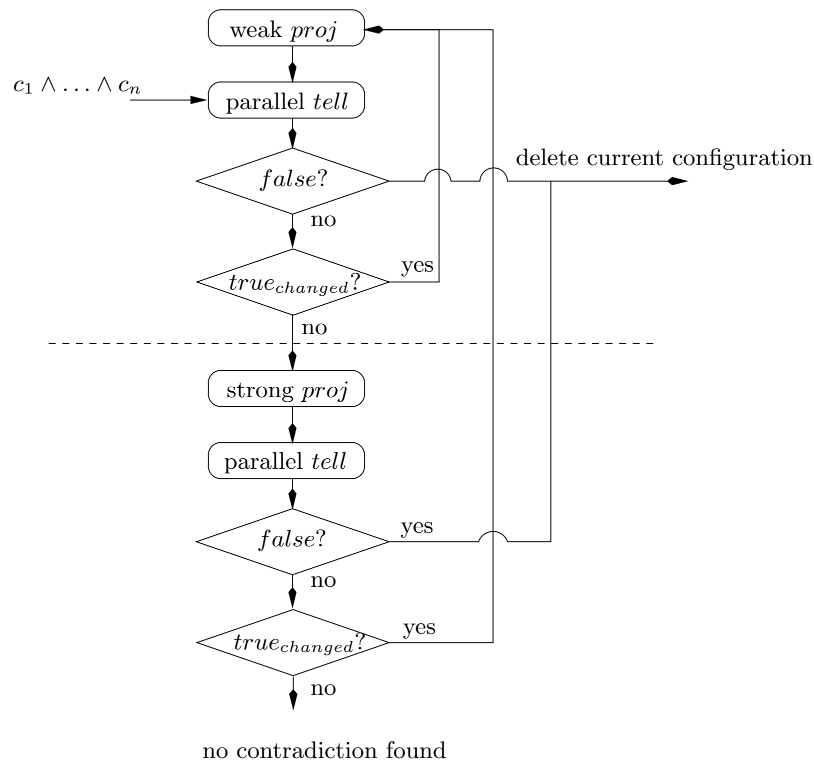



**Fig. 5.** The fixed cooperation strategy

The chosen strategy corresponds to the strategy defined by the reduction relation $\Longrightarrow_{par}$ in [Hof00]. Using this parallel derivation strategy, in one derivation step, first, as many constraints of the constraint pool of one configuration as pos-

sible are propagated, afterwards a projection of the respective constraint stores follows.

Consider Fig.5. Given a conjunction $C = c_1 \wedge \ldots \wedge c_n$ of constraints to be solved, first, a 'parallel *tell*' of all constraints of the pool occurs. If all propagations have been successful and at least one result has been $true_{changed}$, then a projection of all concerned constraint stores follows, where CSPLIB performs a weak projection. If at least one propagation has failed, then the computation (for the current configuration) is aborted. It may happen that there are further configurations to regard for which we start in this case with a parallel *tell* of the constraints of the pool. If for every configuration, a propagation has failed, then we found out, that the initially given constraint conjunction is unsatisfiable. This procedure is repeated until no more changes of the stores occur, i.e. until no successful propagation has had result $true_{changed}$. The upper part of the chart up to the dashed line illustrates this order of events.

The lower part of the chart describes a similar procedure. However, instead of the weak projection which only produces constraint conjunctions, a strong projection is performed here, which is allowed to produce a disjunctive constraint.

The projection of a disjunction causes the splitting of a configuration into an overall configuration in theory and this is realized, as already stated, via creating choice points, saving the current computation state, and a following depth first search. That is, we continue working on one configuration.

As before, the constraints of the constraint pool, among them the newly projected ones, are propagated using the interface functions $tell_\nu$, $\nu \in L$, of the solvers. As in the upper loop, we distinguish three cases:

If no propagation of the newly projected constraints has failed and no successful propagation has supplied result $true_{changed}$, then the computation stops. The system is not able to find a contradiction in the initially given constraint conjunction.

If at least one propagation of the newly projected constraints has failed, then the computation for the current configuration is aborted. If there are further configurations to look at, then we start for one of them with a parallel *tell*. If for every configuration, a propagation has failed, then we found out, that the initially given constraint conjunction is unsatisfiable.

If no propagation has failed and there have been successful propagations with result $true_{changed}$, then the system once again enters the upper loop and continues by weak projections.

As already mentioned, there is the possibility of some variation of the strategy for experiments. This concerns the order of propagation of constraints from the pool. In this way, it is possible to prefer constraints of the form $x = v$ for propagation, where $x$ is a variable and $v$ is a value which express variable bindings. This is based on the idea that the propagation of bindings may, in general, restrict the set of solutions more than other constraints do, and that their propagation may be of lower cost than that of other constraints. As well, it is possible to fix the order of constraint systems propagating their constraints next.

### 4.3 System Configuration – Examples

The implemented overall system can be configured by the user. This concerns the constraint solvers to be integrated as well as strategy aspects and further parameters to support the evaluation of experiments.

These configuration items are given together with the conjunction of constraints to be solved in a file. This file consists, in general, of four parts:

- an (optional) [**global**] part which contains the predefinition of strategy parameters, and parameters to support the evaluation of experiments.
- a part [**solver**], where the constraint solvers to be integrated are specified.
- an (optional) [**configuration**] part for configuring the signature of the solvers.
- a part [**constraints**] which contains the constraints to be solved.

Now, for illustration we will have a look at two example files. For a detailed description see [GS01]. Consider first the input file in Fig.6 which corresponds to Example 1.

```
[global]
  verbose = 1
  outvars = {K}
  interactive = no
[solver]
  CS1 = solver.brandeis.Brandeis with config1
  CS2 = solver.brandeis.Brandeis with config2
  CS3 = solver.csplib.CSPlib with config3
[config1]
 functions = {+,-,*,/,^}
 relations = {=,<=,>=,>,<,in}
[config2]
 functions = {exp}
 relations = {=,<=,>=,>,<}
[config3]
 relations = {=,inn -> in}
[constraints]
  R1 = 100000;
  R2 in [100000, 4.0E5];
  ( 1 / R ) = ( 1 / R1 ) + ( 1 / R2 );
  VK = V * (1 - exp(- (t) / (R * K)));
  VK = 0.99 * V;
  t in [0.5, 1.0];
  K inn {1.0E-6, 2.5E-6, 5.0E-6, 1.0E-5, 2.0E-5, 5.0E-5};
```

**Fig. 6.** Input file: The electric circuit

In the part `[global]`, the determination `verbose = 1` fixes the verbose level which is used to control the amount of output during the computation. The set of variables w.r.t. which a projection of all constraint stores is shown to the user when the system has computed a normal form, is fixed by the set `outvars`. In our

case, we are interested in the value of the capacitor `K`. The parameter `interactive` controls the continuation of the computation after having computed the first set of solutions, i.e. the normal form of the first configuration in the tree of configurations created during the computation. The computation stops waiting for the decision of the user whether a continuation is wished if `interactive = yes` holds, otherwise (`interactive = no`), the computation continues without interrupt.

In the second part, the `[solver]` part, it is stated that we will use the interval solver `CS1`, the finite domain constraint solver `CS3`, and a solver `CS2` which reasons over specific functions, like `exp`, which is simulated by the interval solver with restricted signature. The solvers have been introduced in Sect.4.1. The signature of the finite domain constraint solver is restricted according to the configuration `[config3]`. This configuration fixes the set of predicate symbols of the solver to contain besides `=` only `inn`, which is mapped to the predicate symbol `in`. This has been necessary because the signature of the interval solver contains the symbol `in` as well, but with a different interpretation of the predicate.

The forth part `[constraints]` of the file gives the constraints to be solved according to Example 1. The first three constraints, the fifth and the sixth constraints are constraints of `CS1`, the interval solver. The constraint `VK = V * (1 - exp(- (t) / (R * K)))` is mixed over the symbols of the solvers `CS1` and `CS2` and it must be transformed by the system according to Example 2 into a conjunction of pure constraints. The last constraint can be assigned to the finite domain solver `CS3`. Obviously, to narrow the solution space of the entire constraint conjunction we need the cooperation of `CS1`, `CS2`, and `CS3` here, none of the single solvers is able to handle all given constraints.

Consider as a second example the well known cryptoarithmetic problem of SEND-MORE-MONEY. It is given by the arithmetic equation on the left hand side of Fig.7. It is asked for solutions such that each letter represents a different digit and the given equation holds. On the right hand side of Fig.7 we have given the only solution for which M = 1 holds. An input file for this problem is given in Fig.8. Besides the already considered parameters, in the `[global]` part of this file

| The problem: | | A solution: | |
|---|---|---|---|
| | S E N D | | 9 5 6 7 |
| | + M O R E | | + 1 0 8 5 |
| | M O N E Y | | 1 0 6 5 2 |

**Fig. 7.** The SEND-MORE-MONEY problem

the parameter `pool` is determined. It is fixed by `meta.pool.SolverSorted` which schedules the order of constraint solvers whose constraints are to be propagated next as given in the second part `[solver]`, i.e. the finite domain constraint solver has the highest priority.

We use in the cooperation a further constraint solver `AllDiff`. This solver is able to handle the predicate symbol `alldiff`. Having received the constraint `alldiff(S, E, N, D, M, O, R, Y)`, it projects the conjunction S≠E ∧ S≠N ∧ ... ∧ O≠Y ∧ R≠Y of disequations between every two argument variables. The implementation of this solver is a realization of a general method to extend black

```
[global]
  verbose     = 1
  interactive = yes
  outvars     = {S, E, N, D, M, O, R, Y}
  pool        = meta.pool.SolverSorted
[solver]
  CSPlib  = solver.csplib.CSPlib
  LinAr   = solver.linar.LinAr
  AllDiff = solver.allDiff.AllDiff
[constraints]
  alldiff(S, E, N, D, M, O, R, Y);
  S, E, N, D, M, O, R, Y, U1, U2, U3 in {0 .. 9};
       D + E = Y + (U1 * 10);
  U1 + N + R = E + (U2 * 10);
  U2 + E + O = N + (U3 * 10);
  U3 + S + M = O + ( M * 10);
```

**Fig. 8.** Input file: SEND-MORE-MONEY

box constraint solvers by new predicates without changing the solver directly and after their integration into the combined architecture. The method is more elaborately described in [Hof01]. The extension of a solver, in this case `CSPlib` as well as `LinAr`, with a new predicate is done by integrating a new solver which simply represents a translation mechanism into the overall system.

The signatures of the incorporated solvers are not restricted. The forth part of the file gives one possible representation of the problem by constraints. The constraint `alldiff(S, E, N, D, M, O, R, Y)` can be assigned to the solver `AllDiff`, the second constraint is a finite domain constraint, i.e. a `CSPlib` constraint. All other constraints are linear constraints over rational numbers and can thus be handled by `LinAr`. As in the previous example, to narrow the solution space of the entire constraint conjunction we need the cooperation of all three solvers.

In the above two examples, we have shown some possibilities of system configuration according to the current requirements. There are further predefined configuration items. The system is allowed to be reconfigured by changing the configuration items in the input file without a following recompilation. It allows to integrate further constraint solvers with a suitable interface definition and to extend the configuration items in a simple way.

## 5 Conclusion

This paper describes a prototypic implementation of a system of cooperating constraint solvers. After a short reintroduction of the theoretical description of the system of [Hof00,Hof01] in Sect.3, we described its prototypic implementation in Sect.4.

Three freely available constraint solvers which have been chosen for their integration into the overall system are shortly introduced followed by a description of

the implemented control mechanism and examples of input files. The prototypic implemented overall system can be configured by the user w.r.t. the constraint solvers to be integrated, strategy parameters, and further parameters for support of experiments. The predefined set of configuration items can be extended by the user. The implementation allows to integrate further solvers with a suitable interface definition. The configuration of the system as shown in Sect.4.3 by examples is easy and comfortable. The system has been tested with examples (for diagnosis of electric circuits) up to a magnitude of 100 variables and constraints, which can be handled in an adequate time of some seconds. The system is freely available.

The implementation is due to a diploma thesis [GS01]. There have been strict time limitations and, thus, the current system is very preliminary and error prone and should be considered with much care. Its main purpose was to prove the practicability of the theoretical approach. However, a reimplementation is ongoing. At this, we will take the possibility of a more flexible handling of strategies into consideration. An evaluation of experiments and a more detailed theoretical treatment of the subject of strategies is, as well, a topic of current research.

The prototypic implementation has confirmed the theoretical design of the overall system. The architecture of the overall system as well as the theoretical description of the uniform interface for constraint solvers have been shown to be expedient for an actual implementation.

## References


[Bra01] The Brandeis Interval Arithmetic Constraint Solver, January 2001. Available from `http://www.cs.brandeis.edu/~tim/`.

[CSP01] A 'C' Library of Routines for Solving Binary Constraint Satisfaction Problems, January 2001. Available from `http://www.ai.uwaterloo.ca/~vanbeek/software/csplib.tar.gz`.

[GS01] E. Godehardt and D. Seifert. Kooperation und Koordination von Constraint Solvern – Implementierung eines Prototyps. Master's thesis, University of Technology Berlin, 2001. (in German), Available from `http://uebb.cs.tu-berlin.de/~ph/DA.SeifertGodehardt/`.

[Hof00] P. Hofstedt. Better Communication for Tighter Cooperation. In J. Lloyd, editor, *First International Conference on Computational Logic – CL'00*, volume 1861 of *LNCS*. Springer-Verlag, 2000.

[Hof01] P. Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Dresden University of Technology, 2001.

[Hon94] H. Hong. Confluency of Cooperative Constraint Solvers. Technical Report 94-08, Research Institute for Symbolic Computation, Linz, Austria, 1994.

[Krzan] O. Krzikalla. Constraint Solver für lineare Constraints über reellen Zahlen. Großer Beleg. Technische Universität Dresden, 1997. (in German).

[Mon96] E. Monfroy. *Solver Collaboration for Constraint Logic Programming*. PhD thesis, Centre de Recherche en Informatique de Nancy. INRIA-Lorraine, 1996.

[Rue95] M. Rueher. An Architecture for Cooperating Constraint Solvers on Reals. In A. Podelski, editor, *Constraint Programming: Basics and Trends.*, volume 910 of *LNCS*. Springer-Verlag, 1995.