A Distributed Cooperating Constraints Solving System

(*International Journal on Artificial Intelligence Tools*, vol. 4, n° 1–2, pp 93–113,
World Scientific, June 1995)

Philippe Marti
marti@unice.fr

Michel Rueher
rueher@essi.fr

I3S-CNRS, Université de Nice – Sophia Antipolis, Route des Colles, B.P. 145,
06903 Sophia Antipolis Cedex, FRANCE

Abstract. An appropriate combination of *symbolic and numeric solvers* often makes it possible to solve problems that none of these solvers can tackle alone. In this paper, we specify a cooperative architecture which allows *using concurrently heterogeneous solvers* when handling constraints over the reals. This architecture is based upon agents that communicate via *asynchronous message passing.* Agents are synchronized when a failure or a success occurs. Disjunctive constraints are handled by backtracking. Operational semantics and terminating conditions of such systems are discussed. Implementation issues are addressed. We end the presentation by several examples and give some computational results from a first prototype.

1 Introduction

1.1 Motivation

Many industrial applications can be modeled by a set of constraints defined over the reals (e.g., financial applications, thermal flow problems, electro-mechanical engineering problems). These systems of constraints are usually non-linear or even non-polynomial, and therefore they cannot be tackled by existing constraints solvers: *CLP* languages [JM94] require the constraints to be linear, computer algebra systems [GCL91] cannot solve systems of non-polynomial equations and inequalities[1], numeric solving techniques —such as *interval propagation* [Ben95]— allow one to narrow the domains of the variables for any system of constraints over the reals but in the general case they can neither compute the range of values of the variables nor detect whether a system of constraints is inconsistent or not[2].

However, a careful examination of the particular applications often shows that an appropriate combination of symbolic and numerical techniques allows one to compute the solutions expected by the end user. For instance, let $S1$ be the following system of constraints:

$$x^2 - y^2 = 0 \tag{1}$$
$$x + y < 25 \tag{2}$$
$$x + y > 0 \tag{3}$$
$$-100 \le x \le 100 \tag{4}$$

[1] The Collins' method [Col75] used by RISC-CLP [Hon93] makes it possible to solve polynomial systems of equations and inequalities. However, the high complexity of the proposed algorithms confines their usefulness to very small (albeit interesting) problems.

[2] The interval calculus is usually unable to narrow the intervals to the range of values of the variables, especially when one variable occurs more than once in a constraint [Moo66].

Neither a solver based on the simplex algorithm, nor a solver based on the Groebner basis algorithm can solve the whole system. Interval propagation techniques —like the 3-Bconsistency algorithm [Lho93]— can only narrow y to $[-100, 100]$. Nevertheless, a solver based on the Groebner basis algorithm can simplify equation (1) and generate the following disjunctive constraints:

$$x = y \tag{5}$$

$$x = -y \tag{6}$$

Then, the simplex algorithm trivially detects the inconsistency of $\{2, 3, 4, 6\}$ and it can compute from $\{2, 3, 4, 5\}$ the upper and lower bounds of x and y, i.e., it will determine that x and y are inside the interval $[0, \frac{25}{2}]$.

1.2 Aims of the paper and related work

To provide support for the automatic sequencing of different constraint solvers, we have proposed an approach to *solve systems of constraints over real numbers* based on *cooperation* between solvers [MR94, Rue95]. The basic ideas of our approach are :

- To allow such a cooperation, the system of constraints is split into different subsets of constraints according to the capabilities of the available solvers;
- Each solver manages its own constraint store[3] and has its own solving strategies.

The architecture of our system is based upon the concept of *agents* [Hew77, Agh86] that communicate via *asynchronous message passing*. Messages propagate failures as well as new derived constraints.

Our model is inspired from works on *Concurrent Constraint Logic Programming languages (cc)* which are based on communication and synchronization by *Ask* and *Tell* messages (e.g., [Sar93, MRS94]). The originality of our approach lays in our objective to mix *heterogeneous solvers* to tackle systems of constraints over the reals whereas previously mentioned works aim at giving a general framework for processing constraints concurrently with homogeneous solvers. In our model, nondeterminism is handled by backtracking and *synchronization* of all solvers is done after each successful or failing resolution, while in most *cc* systems the agents are duplicated. Of course, when a choice point occurs one could also consider processing the various alternatives concurrently. From an operational point of view, this solution is nevertheless unrealistic due to the resources that the solvers require and the huge amount of disjunctive constraints that may appear in numerous cases.

The focus of this paper is on the effective implementation of a cooperative constraint system aimed at solving constraints over the reals. In a first step we detail the communication process when several non-deterministic solvers are involved in the cooperation. An operational prototype —based on existing solvers such as PrologIII, MapleV and Interlog — is introduced. In a second step, we provide some guidelines for the extension of our system, and show how new solvers can be integrated.

[3] One should note that the same constraint may be dispatched to different solvers. Local stores do therefore not represent a distributed store as in Janus [SKL90].

Nelson and Oppen [NO79] first proposed a method for combining decision procedures for several theories into a single decision procedure. In their system, each satisfiability procedure deduces and propagates to the other satisfiability procedures all equalities between variables entailed by the conjunction it is considering. The equalities propagation procedure is based upon an algorithm which *sequentially* applies the different solvers and which handles successively the choices points (i.e., the disjunctions).

Cooperative problem solving has also been investigated in the field of artificial intelligence. For instance, in [DLC89] the authors study how a *loosely coupled network* of problem solvers can work together to solve problems that are beyond their individual capabilities. In their approach, cooperative nodes can use local expertise, resources, and information to individually solve different sub-parts of an overall problem. However, their focus is on very general issues (e.g., cognitive modeling of negotiation) and no formalization of the resolution process is proposed. More recently, Khedro and Genesereth [KG94] have modeled multi-agent cooperation as a distributed constraint satisfaction problem. Agents independently solve distributed sets of constraints defined over distributed sets of objects. Their architecture is also based upon a *negotiation strategy* which aims at minimizing backtracking.

Colmerauer [Col92] has proposed a "naive and incomplete algorithm" for solving systems of non-linear constraints. The initial system of constraints S is split into two subsystems, a linear one and a non-linear one which only contains constraints of the form $z = x \times y$, where x and y are variables. The algorithm repeatedly applies the following actions:

1. Solving the linear part of S,
2. Considering the equations $x = constant$ that are entailed by the linear part of S and replacing each variable x by $constant$ in the right hand sides of non-linear equations.

This algorithm is implemented in PrologIII by using delay mechanisms. In the same spirit, a more elaborated coupling of a quadratic and a linear solver has recently been implemented in QUAD-CLP(R) [PB94].

Quite recently, Hoon Hong [Hon94] has proposed a scheme for the design of a cooperative constraint solver. He suggests repeatedly applying each sub-solver until no change occurs. His paper is devoted to the study of confluence of such algorithms but does not explicitly address the problem of communication between the sub-solvers.

In a forthcoming paper, Beringer and De Backer [BB95] propose a tight coupling between two deterministic solvers: a domain reduction solver and a real linear solver. The domain reduction solver signals each bound it has inferred for variables which are also included in some real linear constraints, while the linear solver sends to the domain reduction solver the values of shared variables as soon as they are fixed. To implement efficiently the cooperation, the authors have designed a specific domain reduction algorithm and a "'revised" simplex algorithm. Since their goal is the design of a new language (i.e., ICE), the emphasis is rather on the integration of the algorithms than on the cooperation between distinct solvers.

1.3 Layout of the paper

Section 2 recalls the main points of our general scheme for (heterogeneous) cooperating solvers on reals. Agents and messages are formally defined. The operational semantics and terminating conditions are studied in section 3. Section 4 is devoted to the definition of the message handling process. The architecture of an operational prototype is introduced in section 5. In section 6, several examples are provided to illustrate the capabilities of our system, and some computational results from a first prototype are given.

2 A general scheme for cooperation between solvers on reals

2.1 Overall Architecture

In our system, a solver is encapsulated in an agent, i.e., a process endowed with some solving capabilities and its own data ("local store"). An interface module —in charge of the communication— is associated with each agent. Interface modules go beyond simple data transcoding and store management: they analyze the messages and they can schedule the work of the solver according to some heuristics. For instance, they can select the message to be processed according to the "first fail" heuristic when the "composition" of all available messages is commutative.

The resolution process starts by a decomposition of the initial system of constraints and by sending subsets of constraints to the solvers (according to their capabilities). Each agent tells the other ones about its relevant results (e.g., instantiations or simplifications). Of course, results sent back by a solver should neither be equivalent to the constraints it has just received, nor be deductible from the previous value of its local store. A terminal state is a state where each solver has processed all its messages. The termination of the resolution can be guaranteed if the resolution process of each solver halts, and if the propagated constraints do not contain any new variables.

Agents do not communicate directly one with each other. Instead they only communicate —by means of messages— with a specific agent called the *monitor* which supervises the resolution. The monitor is in charge of the decomposition of the initial system of constraints and of the detection of a terminal state. It also collects the messages sent by the *solver* agents and forwards constraints only to the agents able to process them. This way of processing the messages has been chosen instead of a "broadcasting" mode *to simplify the processing of disjunctive constraints*. Indeed, the resolution of non-linear constraints is always likely to generate disjunctive constraints which lead to the definition of choice points. When a solver retrieves disjunctive constraints, the monitor agent asks the *solver* agents to generate choice points. Conceptually, making a choice point corresponds to a copying of the current value of the local store. The successive values of the local store are stacked[4] up at the level of each solver. The different disjuncts of such constraints will be sequentially processed. One should note that the management of the disjunctive constraints at the monitor level entails that the messages sent to the solvers can only carry conjunctions of elementary constraints.

The result of a successful resolution corresponds to the conjunction of the constraints held in all the local stores. The degree of consistency of this result is the strongest consistency degree guaranteed by the solvers that work on the whole set of constraints. For instance, if the resolution of a constraints system involves a solver based on the simplex algorithm and a solver based on interval propagation, and if the resolution succeeds, one can only conclude

[4] On the operational level, one can avoid copying the whole context by using more efficient mechanisms when the solvers have backtracking capabilities.

that *if the initial system has solutions, the values of the non-instantiated variables are inside of the computed intervals.* Of course, if no solver works on the whole set of constraints, nothing can be said about the degree of consistency of the result. This corresponds to the situation where no solver works on the whole set of constraints.

When a failure occurs, all the solvers should backtrack and restore the appropriate local store and messages. However, as the messages are processed concurrently, all solvers have not necessarily generated the same choice points when a failure occurs. Thus, a systematic backtrack to the last choice point would lead to a non-coherent configuration and message interleaving problems. To avoid such problems, backtracking has to be based on labeling of messages and local stores, while the restoration of already processed messages must be achieved at the level of the monitor.

The relationships between the different components of a cooperating constraints system are sketched in figure 1. As shown, we assume that there is an additional agent —named *user interface*— which communicates with the end user. This agent is in charge of sending the initial set of constraints to the monitor and it displays the results according to information provided both by the monitor and the local stores of the solvers.

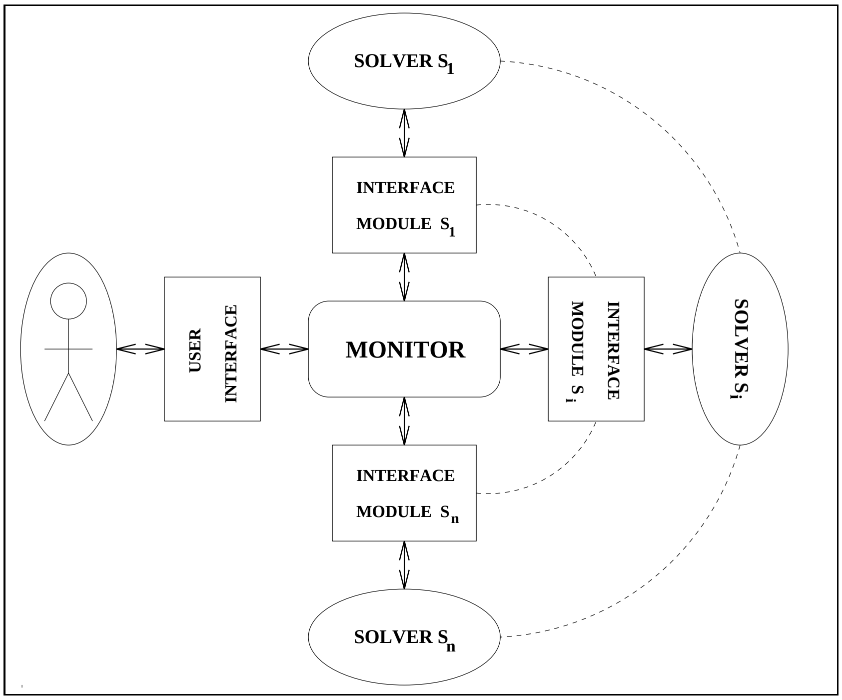

Fig. 1. Overall architecture

2.2 Agents and Messages

We now specify more formally the interactions between the solvers and the monitor. In the following we assume that Γ, the set of the available solvers is restricted to:

- $\mathcal{A}_{\mathcal{L}}$, the set of the solvers that are able to handle linear equalities and inequalities,
- $\mathcal{A}_{\mathcal{P}}$, the set of the solvers that are able to handle polynomial equalities,
- $\mathcal{A}_{\mathcal{NP}}$, the set of the solvers that are able to handle systems of non-polynomial equalities and inequalities.

We use tuples to represent successive states and messages queues. A tuple made of the elements $L_1, ..., L_n$ is noted $< L_1, ..., L_n >$. The empty tuple is noted by $<>$ while $L1 \cdot < e > \cdot L_2$ denotes the concatenation of the tuple L_1 with the tuple made of one element e and the tuple L_2. We use the notation $e \in L$ to mean that e is an element of the tuple L.

We also use the following notations, possibly subscripted:

- a, b, c denote real constant numbers,
- x, y, z denote real variables, '_' denotes the anonymous variables which are assumed to be existentially quantified,
- c denotes an "elementary" constraint over the reals (i.e, an equation, a disequation or an inequality over the reals)
- $C = \bigwedge_{j=1}^{n} c_j$ represents a conjunction of elementary constraints; we sometimes use the notation $c \in C$ to mean that c is a conjunct of C,
- $D = \bigvee_{i=1}^{n} C_i$ represents a disjunctive constraint,
- S_{olver} is any solver agent, M_{onitor} is the monitor agent, A denotes any agent.

We denote by $S \overset{S_{olver}}{\vdash} c$ the entailment of c from the current value S of the local store of S_{olver}. $S \overset{S_{olver}}{\vdash} \perp$ means that S is not consistent.

Definition 1 (solver and monitor agents)

An *agent* is associated with each solver in Γ. A solver *agent* is defined by a relation $S_{olver}(\Pi, F, M)$ where:

- S_{olver} is the name of the agent,
- Π is a stack containing the values of the local store which are required for backtracking. The current value (noted S_π in the following) corresponds to the top of Π,
- F is the state flag of the agent:
  - $F = success$ if the last constraint C processed by the agent was consistent with respect to the current value of its local store,
    i.e, $\neg(S_\pi \wedge C \overset{S_{olver}}{\vdash} \perp)$
  - $F = fail$ if the last constraint C processed by the agent was not consistent with respect to the current value of its local store,
    i.e, $S_\pi \wedge C \overset{S_{olver}}{\vdash} \perp$
- M is the queue of messages that S_{olver} has to process.

The monitor is a specific agent defined by the relation $M_{onitor}(F, M, L_D)$ where M has the same meanings as above, and where L_D is a queue of disjunctive constraints to be handled. F is a state flag whose value is *success* if the value of the state flag of all the solver agents is *success*; otherwise $F = fail$.

The configuration or *current state* Σ_A of an agent A is defined by the following information:

- the state flag,
- the values of the local store,
- the queue of messages to be processed.

We note respectively $S_\pi(A)$, $flag(A)$ and $Mess(A)$ the current value of the local store, the flag and the queue of messages of an agent A.

Definition 2 (message)

A `message` can have one of the following forms:

- `Tell`$(M_{onitor} \to S_{olver}, C)$,
- `Tell`$(S_{olver} \to M_{onitor}, D)$,
- `M_Choice`$(M_{onitor} \to S_{olver})$,
- `Restore`$(M_{onitor} \to S_{olver})$,
- `Fail`$(S_{olver} \to M_{onitor})$,

A `Tell` message is sent by the monitor agent to the solvers in order to transmit them the constraints they have to process. A `Tell` message is sent by a solver to monitor in order to forward the result of the resolution[5]. A `M_Choice` message is sent by the monitor agent to the solvers in order to notify them to create a choice point. A `Restore` message is sent by the monitor agent to the solvers to ask them to backtrack and to restore the appropriate local store (according to the state labeling). A `Fail` message is sent to the monitor agent by a solver if the latter has detected an inconsistency while trying to add a constraint to its local store.

3 Operational Semantics

In our cooperative architecture, each state of the computation consists of the set of all current states associated with the different agents. Each computation step will model the evolution of an agent. Such an evolution will be performed by applying a transition rule (i.e, the processing of a message by the agent).

Definition 3 (transition rule)

For the agent A, the processing of the messages corresponds to a sequence of *inference steps* defined by a transition rule r of the form:

$$r : \Sigma_A^i \overset{r(A)}{\to} \Sigma_A^{i+1}, \mathcal{M}$$

where:

[5] For generalization's sake, one considers this result as a disjunction of constraints even though in numerous cases the result will be a conjunction of constraints.

- Σ_A^i is the current state of agent A before transition r,
- Σ_A^{i+1} is the state of agent A after transition r,
- $\mathcal{M}$ is the queue, possibly empty, of the messages sent by A after commitment of the transition r.

Definition 4 (computational state)

A computational state consists of the set of current states of all agents. For instance, the initial computational state is defined by:
$\{\forall A_l \in \mathcal{A_L} \,.\, A_l(<>, success, <>),\ \forall A_p \in \mathcal{A_P} \,.\, A_p(<>, success, <>),$
$\quad \forall A_{np} \in \mathcal{A_{NP}} \,.\, A_{np}(<>, success, <>),\ M_{onitor}(success, <>, <>)\}$

Definition 5 (computational step)

Let $\Sigma^1 = \{\Sigma^1_{A_i} | A_i \in \mathcal{A_L} \cup \mathcal{A_P} \cup \mathcal{A_{NP}} \cup \{M_{onitor}\}\}$ be a computational state, and r a transition rule which enables the evolution of agent A_j. Then, the application of r to Σ^1 is a computational step which yields a new computational state $\Sigma^2 = \Sigma^1 \backslash \{\Sigma^1_{A_j}\} \cup \{\Sigma^2_{A_j}\}$.

Definition 6 (successful computational state, failing computational state)

A successful computational state is a state Σ where the monitor and each solver are in a successful state and where their queues of messages are void, i.e.,
$\quad \forall \Sigma_A \in \Sigma.(flag(A) = success) \wedge (Mess(A) = <>)$
The result of the computation corresponds to the *conjunction* of the constraints in the current local stores of all the solvers.

A failing state is a state where at least one solver is in a failing state, i.e.,
$\quad \exists \Sigma_A \in \Sigma.(flag(A) = fail)$

Definition 7 (terminal computational state)

A terminal computational state is a state Σ where no more transition rules can be applied, i.e., a successful or failing state where no more backtracking can be performed. Thus, a terminal state is characterized by one of the following conditions:

- Every solver agent has processed successfully its messages and the monitor's stack of disjunctive constraints is empty, i.e.,
  $(\forall \Sigma_A \in \Sigma.(flag(A) = success) \wedge (Mess(A) = <>)) \ \wedge\ M_{onitor}(success, <>, <>) \in \Sigma$,
- The monitor is in a definitive failure state, i.e.,
  $\exists \Sigma_A \in \Sigma.\Sigma_A = M_{onitor}(fail, <>, <>)$.

To warrant termination of the resolution process, the following conditions are required:

1. The constraints sent back by a solver S_{olver} should neither be equivalent to a subset of the ones that it has just received, nor be deductible from the previous value of its local store. Thus, we define the new inferred constraints (when a state transition occurs) as a disjunction of constraints of S'_π which do not occur in C and which cannot be entailed by S_π.
   $\Delta(S'_\pi, S_\pi) \setminus C = D$ such that:
   $$\forall C_i \in D \,.\, C_i = \bigwedge c_j \mid ((S'_\pi \overset{S_{olver}}{\vdash} c_j) \ \wedge\ \neg(S_\pi \overset{S_{olver}}{\vdash} c_j) \ \wedge (c_j \notin C))$$
   where S_π is the top of $\Pi_{S_{olver}}$ and S'_π is its new value after the processing of C by S_{olver},

2. The constraints sent back by a solver should not contain any new variables:
   $var(\Delta(S'_\pi, S_\pi) \setminus C) \subseteq \{var(S_\pi) \wedge var(C)\}$
   where $var(S_\pi)$ (resp. $var(C)$) is the set of variables of S_π (resp. C).

Most of the available constraints solvers have the following properties:

1. At the end of each resolution, a projection of the new inferred constraints on the initial set of variables is computed;
2. The new inferred constraints are displayed in a kind of normal form.

If the above conditions hold, all the local entailment tests can be performed syntactically.

Proof of termination

To prove that the resolution of a cooperating solvers system always terminates two additional conditions must hold:

1. The resolution process must be guaranteed to halt for each solver,
2. Numeric solvers have to work on a finite set of numbers[6].

The above conditions and the restriction of forwarded constraints to relations defined on the initial set of variables guarantee that the set of messages is finite. The definition of $\Delta(S'_\pi, S_\pi) \setminus C$ —based upon local entailment tests— ensures that a solver will not forward twice the same message, and hence, that no cycle will occur in the resolution process.

Definition 8 (class of problems)

The class of problems which can be tackled by a cooperating solvers system subsumes the classes of problems that can be individually tackled by the different solvers involved in the cooperation. Indeed, the first transition applied to a solver yields exactly the same result as the one this solver would produce when handling the relevant subset of the initial set of constraints. As the different agents can work concurrently, computing these first solutions with a cooperating solvers system will not entail any significant overhead.

As long as no failure occurs, each computational step enforces the constraints system of one of the local store by including some propagated constraints. As all these constraints are deduced by one of the solvers from a subset of the initial set of constraints, these "tell operations" are sound. Thus, the class of problems which can be tackled by a cooperating solvers system can informally be defined by the constraints systems which can be transformed —by repeatedly applying the transition rules— into a constraints system which is either in a solved form[7] or which contains a sub-system whose inconsistency can be detected at least by one of the solvers of Γ.

More formally, let $\mathcal{R} = \{\Sigma^1, \Sigma^2, ..., \Sigma^n\}$ be the successive computational states yield by a cooperating solvers system when processing a system of constraints D, then the following properties hold :

[6] This condition just requires that numeric solvers work on an approximation of real numbers by "machine interval", e.g., fixed length floating point numbers equipped with safe rounding operations.
[7] A solved form is a simplified form (canonical form or expected end user form).

- If Σ^n is a failing computational state and no Σ^i in $\mathcal{R}$ corresponds to a successful computational state, then D is inconsistent, i.e., if a definitive failure occurs before any solution is exhibited, then $D \vdash \bot$,
- If Σ^n is a successful computational state, then the solution $S = \bigwedge_{A_i \in \Gamma} S_\pi(A_i)$ is an approximation which is at least as accurate than the ones provided by the different solvers involved in the cooperation.

4 Processing of the messages

The way messages have to be processed[8] depends on the number of non-deterministic solvers (solvers which may generate disjunctive constraints) involved in the cooperation. In the general model we consider that any disjunction of constraints satisfying the termination conditions (see definition 7) can be sent by the solvers to the monitor, and that several non-deterministic solvers may be involved in the cooperation. Message processing by the solvers is assumed to be sequential, i.e., the messages queues of the solvers are handled according to the FIFO rule.

When several non-deterministic solvers are involved in the cooperation, a complete exploration of the search tree will require to restore both previous states of the solvers and some already processed messages. While the management of the local stores can easily be realized at the level of each solver, the restoration of already processed messages would be tricky to achieve locally. Thus, we propose to store at the level of the monitor all messages dispatched to the solvers. When a failure occurs, some messages dispatched since the set up of the corresponding choice point have just to be reinserted in the queue of the messages M of the monitor.

In the following we assume that messages, values of a local store Π, and disjunctions in the queue L_D of the monitor can be identified unambiguously through their labels. We also assume that labels hold all relevant path information required for defining matching operation between labels.We note respectively $label(S_\pi)$, $label(m)$ and $label(D)$ the labels of local store value S_π, message m and disjunction D.

4.1 Initialization

Let $D = \bigvee_{i=1}^{m} C_i$ be the initial set of constraints. The resolution process starts by sending to the monitor the message $\mathtt{Tell}(I_{nit} \rightarrow M_{onitor}, D)$.
The monitor agent builds a partition of the constraint C_1 into $part(C_1) = (C_{li}, C_{le}, C_{pe}, C_{np})$ where:

1. $C_1 \Leftrightarrow C_{li} \wedge C_{le} \wedge C_{pe} \wedge C_{np}$
2. elements of C_{li} are linear inequalities,
3. elements of C_{le} are linear equalities,
4. elements of C_{pe} are polynomial equalities whose total degree is greater than one,

[8] Problems concerning the protocol for exchanging information are not addressed here. We assume that the communication between agents is based on a reliable communication language (e.g. KQML [FMM94, LF94]) which guarantees that messages to a single destination arrive in the order they were sent, and that messages delivery is reliable. Communication between the monitor and the solvers is assumed to be instantaneous (i.e., transmission of messages takes no time).

5. elements of C_{np} are non-polynomial equalities, (e.g., constraints defined with functions like `log, sin, cos`, etc...). A non-linear inequality is converted to a non-linear equality and a linear inequality thanks to a slack variable.

The different subsets of the decomposition of C_1 are sent to the solvers that are able to handle them. Then, the monitor agent pushes in its stack L_D the remaining constraints of the disjunction.

4.2 Processing of the `Tell` messages by the *solvers* agents

When the head of the messages queue M of a solver A consists in a message $\texttt{Tell}(M_{onitor} \rightarrow A, C)$, the constraint C is handled[9] in the following way:

- if C can be deduced from S_π, then C is ignored,
- if there exists c_i in C such that c_i can be disproved from S_π, then S_π is set to $\perp$ and a `Fail` message is sent to the monitor,
- if the constraint C can neither be deduced nor disproved from S_π by A, then its non-redundant conjuncts c_i are added to the local store and $\Delta(S'_\pi, S_\pi) \setminus C$ is sent to the monitor.

4.3 Processing of the `M_Choice` and the `Restore` messages by the *solver* agents

Processing a `M_Choice` message requires simply that the solver duplicates its current value of the local store. Processing a `Restore` message corresponds to the reverse operation: the appropriate state of the local store has just to be restored.

4.4 Processing of `Tell` messages by the monitor

The monitor processes `Tell` messages in the order of arrival as long as its messages queue M does not hold any `Fail` message. Except this latter condition, processing of `Tell` messages is similar to the processing of the initial message described in section 4.1. For each conjunction C of a constraint carried by a `Tell` message, the monitor builds a partition of C and the different subsets of the decomposition of C are sent to the solvers able to handle them. In case of disjunctive constraints, the monitor asks them to create a choice point before handling the first conjunction and pushes the remaining constraints of the disjunction in L_D.

4.5 Processing of `Fail` messages by the monitor

When the monitor is in a successful computational state and its messages queue M is holding a `Fail` message, this one is immediately taken into account. If the queue L_D does not hold any disjunction whose label matches the one of the `Fail` message, a definitive failure occurs. Otherwise, a `Restore` message is dispatched to all solvers and the monitor's flag is set to $fail$. More formally, processing of a `Fail` message enables the evolution of monitor in the following way:

[9] Keep in mind that `Tell` messages sent back by the monitor to solver agents do not contain disjunctive constraints.

$$\begin{array}{l} Monitor(success, M, L_D) \\ \qquad Fail \downarrow \qquad \text{if } \texttt{Fail} \in M \\ Monitor(fail, M', L'_D), \mathcal{M} \end{array}$$

where the values of M', L'_D and $\mathcal{M}$ depend on the value of M and L_D:

1. If there is no message M_D in L_D such that $label(M_D)$ matches $label(\texttt{Fail})$, then a definitive failure occurs, and thus, M', $\mathcal{M}$ and L'_D are set to $<>$.
2. If $L_D = L_{D_1} \cdot < M_D > \cdot L_{D_2}$ such that $M_D = \texttt{Tell}(S_{olver} \rightarrow M_{onitor}, \bigvee_{i=n}^{m} C_i)$ and the label of M_D matches the one of the Fail message, then $L'_D = L_{D_2}$, $\mathcal{M} = \{\texttt{Restore}(M_{onitor} \rightarrow A) \mid A \in \Gamma\}$ and M' holds:
   - the messages of M which have not been generated with information resulting from the dispatching of message C_{n-1},
   - the messages dispatched by the monitor since the set up of the choice point corresponding to M_D and which have not been generated with information resulting from the dispatching of message C_{n-1}.
   - the message $\texttt{Tell}(S_{olver} \rightarrow M_{onitor}, \bigvee_{i=n}^{m} C_i)$.

   The ordering of all these messages is done according to increasing labels.

As long as the monitor flag is set to $fail$ all messages except `Fail` messages will be ignored. A `Fail` message which occurs when the monitor is in a failing state will only be taken into account if its label is a predecessor of the one of the previous processed `Fail` message. In the latter case, the same processing as in a successful state of the monitor will be performed.

4.6 Backtracking and synchronization of the solvers

In case of success, if the queue of remaining disjunctive constraints is not empty, the monitor dispatches a `Restore` message to the solvers, moves the first message from L_D to M and sets its flag to $Fail$.

Thus, when a failure or a success occurs, the `Restore` messages dispatched by the monitor will both guarantee the restoration of an appropriate local state and entail a *synchronization* of all solvers agents. This synchronization results from the fact that as long as the monitor flag is set to $fail$ only `Fail` messages will be considered. The monitor will switch to a success state as soon as all solvers have processed the `Restore` message successfully.

5 Architecture of the prototype

In order to validate our model, we have implemented a prototype which allows carrying out a linear system solver, a polynomial equality solver and an interval propagation based solver. All inferred constraints are forwarded by these solvers. Among these, only the solver of polynomial equalities can generate choice points. Thus, backtracking just requires the restoration of the appropriate local stores; unlike the general case described above, no previously dispatched messages have to be sent again to the solvers by the monitor.

5.1 Software components

The prototype implementation is composed of the following modules:

- Solvers that simplify constraint systems which are forwarded by the monitor, and that send back their results in a canonical form (in the native syntax);
- Interfacing modules that translate messages exchanged between the solvers and the monitor;
- A monitor that dispatches the messages to the different solvers (via their interfacing modules) and that manages choice points when a disjunctive constraint occurs.

The solvers are respectively based upon the following algorithms:

- A simplex algorithm simplifying linear constraints, and whose results are conjunction of equalities and inequalities;
- An interval propagation algorithm providing a set of narrowed intervals;
- A Groebner basis algorithm simplifying polynomial equalities, and whose results are given in various canonical forms (Horner form, normal form, ...).

To each solver is attached an interface module which performs the following tasks :

- Entailment tests before the integration (or the propagation) of new constraints by the corresponding solver (these tests are done syntactically);
- Translation of the messages of the constraints from the solver's native syntax to a common exchange language (and conversely from this language to the solver's native syntax);
- Tagging the messages to allow the monitor to process the constraints without any analysis;
- Monitoring the solvers and management of backtracking in a way depending on the capabilities of the solver: if the solver has backtracking capabilities, the interface module will send a suitable request, otherwise it will manage the environments itself.

5.2 Implementation

The prototype implementation is based on a parallel virtual machine using the PVM3 [Sun90] library. Thus, the execution of the different agents can either be distributed on a network of processors or emulated on a single processor. Each agent corresponds to a task in the virtual machine while each task corresponds to at least one process.

The syntax of the messages conforms to the *OpenMath* standard, the communication protocol which is supported by numerous computer algebra systems. Thus no translation of the constraints has to be done by the interface module attached to such solvers.

The linear solver used is that of PrologIII [Col90, Rue93]; the polynomial solver is implemented as a library of the computer algebra system Maple V [CGG+91] and the interval propagation based solver is InterlogII [Lho94]). Interfacing modules are mainly written in *lex* and *yacc*. The monitor is implemented in Prolog III (specific predicates have been added to allow calls to the PVM3 library). The monitor uses a configuration file containing all required information about the capabilities of the solvers.

5.3 Extension of the prototype

The integration of a new solver in our cooperating system requires few developments. One just has to write the attached interface module and to update the configuration file of the monitor. In most cases, one should be able to reuse a significant part of the code of existing interfaces when developing a new interface module.

6 Applications and Computational Results

In this section we present several examples which illustrate the capabilities of our system, and give some computational results from a first prototype.

6.1 A ballistics problem

The problem consists in finding the falling point of an object launched in a uniform gravitational field $\vec{g}$, with an initial speed $\vec{V_i}$ which has an incidence α with the ground (see fig. 2). Moreover, the values of the two last variables are not exactly known (due to mechanical tolerances, imprecision of the measures, etc...). So, the solution will be a “spot” covering the possible falling points of the projectile.

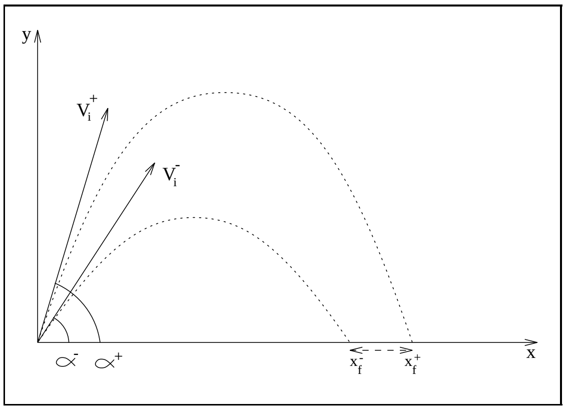

Fig. 2. possible trajectories of the projectile

Modeling of the problem:

The initial speed and incidence of the bullet can be stated as follows:

$$V_y = V_x \tan(\alpha) \tag{7}$$

$$V_x^2 + V_y^2 = V_0 \tag{8}$$

Where V_x and V_y are the xy-coordinates of the initial speed V_i (V_0 is the square of V_i). Both initial speed and incidence are known with uncertainty:

$$\tan(\alpha) \in [0.9, 1.1] \tag{9}$$

$$V_0 \in [500, 501] \tag{10}$$

The equation of the trajectory of a bullet submitted to a constant acceleration $\vec{a}$ is:

$$\vec{M} = \frac{1}{2}\vec{a}t^2 + \vec{V_i}t + \vec{M_0}$$

Where $\vec{M}$ is the vector pointing the bullet, $\vec{a}$ is equal to $\vec{g}$ (the gravitation), $\vec{V_i}$ and $\vec{M_0}$ the initial speed and position of the bullet at time $t = 0$. This equation projected on the xy-coordinates gives:

$$\begin{cases} x = V_x t \\ y = -\frac{1}{2}gt^2 + V_y t \end{cases}$$

We want to know the falling point (x_f, y_f) of the bullet that is, when $t = t_f$:

$$x_f = V_x t_f \quad (11)$$

$$y_f = -\frac{1}{2}gt_f^2 + V_y t_f \quad (12)$$

$$y_f = 0 \quad (13)$$

$$x_f > 0 \quad (14)$$

If we approximate g by $9.81 m/s^2$, the solution of the constraint system is: $x_f \in [50.68, 50.84]$

Resolution:

To handle constraints (7—14) with an interval propagation algorithm, some additional bounds on the variables have to be provided :

$$\begin{cases} V_y = V_x \tan\alpha, & \tan\alpha \in [0.9, 1.1], \\ & V_x \in [0, 10000], \\ V_x^2 + V_y^2 = V_0, & V_y \in [0, 10000], \\ & V_0 \in [500, 501], \\ x_f = V_x t_f, & x_f \in]0, 10000], \\ y_f = -\frac{9.81}{2}t_f^2 + V_y t_f, \; y_f = 0, & \\ t_f \in [0, 10000]. & \end{cases}$$

Then interval propagation can only narrow x_f to $]0, 102.16]$. Which is too large to be considered as an accurate solution.
However, the above equations can be simplified by a Groebner basis algorithm which provides the following solutions :

$$(15)\left\{\begin{array}{l} V_y = V_x(\tan\alpha), y_f = 0, t_f = 0, \\ V_0 = {V_x}^2(\tan\alpha)^2 + {V_x}^2, x_f = 0 \end{array}\right\},$$

$$(16)\left\{\begin{array}{l} V_y = V_x(\tan\alpha), y_f = 0, \\ V0 = {V_x}^2(\tan\alpha)^2 + {V_x}^2, \\ t_f = 0.2038735984\, V_x, \\ x_f = 0.2038735984\, {V_x}^2(\tan\alpha), \end{array}\right\}$$

Now, the interval resolution of (7—15) fails because of $x_f = 0$ but the second system (7–14,16) gives a good approximation of the solution:

$$x_f \in [50.31, 51.16]$$

The whole computation approximately requires five seconds on a Sun 4 computer.

6.2 A generic Problem

Let be the following system of constraints:

$$\forall i \in \{1..n\} \begin{cases} U_i^2 + V_i^2 = 2W_{i-1} \\ R_i^2 \leq -S_i \\ R_i \geq 0 \\ W_0 = 1 \\ W_i \geq U_i, W_i \leq V_i \\ U_i + V_i = 2 \\ X_i + Y_i + Z_i + T_i = W_i \\ X_i + Y_i + Z_i - T_i = 2W_i \\ X_i + Y_i - Z_i + T_i = 3W_i \\ X_i - Y_i + Z_i + T_i = 4W_i \\ -X_i + Y_i + Z_i + T_i = S_i \end{cases}$$

The problem consists in finding the domains of the variables R_i which satisfy the whole system of constraints. None of the three solvers used in the prototype can tackle such a system of constraints alone. However, the prototype of our cooperating constraint solvers system could narrow the domains of all the variables R_i to the interval $[0, 2.64575131106464]$ which is a very good approximation of their range of values (i.e., $[0, \sqrt{7}]$. This result could be obtained even with limited communication between the solvers (i.e., propagation of variables instantiations).

n	1	2	3	4
number of variables	9	18	27	36
number of exchanged messages	14	24	34	46
Time of Simplex Solver	0.02	0.04	0.05	0.12
Time of Groebner Solver	1.05	2.87	8.15	64.35
Time of Interval Solver	3.18	13.17	34.10	52.88

Fig. 3. Table I

The computational results are shown in Table I. Times are CPU times and are given in seconds. On this example a large part of the computational time is spent in the Interval solver due to an asymptotic convergence of the R_i to $\sqrt{7}$. Of course, a less accurate approximation can be computed very much faster. Note that the overhead for communication and messages processing is not significant.

7 Conclusion

Inter-operability is certainly a key issue for progress in constraint solving. The architecture for a cooperating constraints solvers system we have defined in this paper is based on asynchronous communication between heterogeneous solvers. Thus, it enables to *using together both symbolic and numerical solvers* for tackling systems of constraints that none of them could solve alone. Solutions provided by the proposed cooperating solvers system are always at least as accurate than the one which could individually be computed by the different solvers. Moreover, as the latter solutions are computed in the first step of the cooperative resolution, there is no significant overhead for obtaining them.

The prototype that we have developed makes use of various well known solvers. It enables concurrently using a solver of linear equations and inequalities, a solver of polynomial equalities, and a solver based on interval propagation. This prototype allowed us to validate our approach and the first experimental results are promising. Further work concerns the implementation of heuristics to choose the message to be selected first in the queues of messages, and the introduction of redundant constraints in order to improve efficiency of such cooperative systems. For instance, when a conjunction of elementary constraints is split into several sub-systems, shared information could be augmented by defining some "redundant" linear equations: for each non-linear equation, a linear form can be defined by introducing new intermediate variables. Such redundant information could help detecting inconsistencies earlier. In the same way, redundant polynomial equations could also be generated from non-polynomial equations.

Thanks: Thanks to Patrice Boizumault, Olivier Lhomme, Christine Solnon and Dan Vlasie for their helpful comments on earlier drafts of this paper.

References

[Agh86] Gul Agha. *Actors:A model of Concurrent Computation in distributed Systems.* MIT Press, 1986.

[BB95] Henri Beringer and Bruno De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In C. Beierle and L. Plumer, editors, *Logic Programming: formal methods and practical applications.* Elsevier Science Publisher B.V., 1995.

[Ben95] F. Benhamou. *Constraint Programming: Basics and Trends*, chapter *Interval Constraint Logic Programming* , pages 1–21. LNCS 910. Springer Verlag, March 1995.

[CGG+91] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan B.L. Leong, and S.M. Watt al. *Maple V Language Reference Manual.* Springer-Verlag, New York, ISBN 0-387-97622-1, 1991.

[Col75] G.E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. *LNCS*, pages 134–183., 1975.

[Col90] Alain Colmerauer. An introduction to PROLOG-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[Col92] Alain Colmerauer. Résolution naive de contraintes non linéaires. Technical report, Groupe Intelligence Artificielle, Université Aix – Marseille II, March 1992.

[DLC89] E.H. Durfee, V.R. Lesser, and D.D. Corkill. Cooperative distributed problem solving. In E.A. Feigenbaum A. Barr, P.R. Cohen, editor, *The Handbook of Artificial Intelligence*, volume IV, pages 83–147. Addison-Wesley Pub. Co., Reading, MA, 1989.

[FMM94] T. Finin, D. McKay, and R. McEntire. KQML - A Language and Protocol for Knowledge and Information Exchange. Technical Report CS-94-02, Computer Science Department, University of Maryland, UMBC, Baltimore MD 21228, 1994.

[GCL91] K. O. Geddes, S. R. Czapor, and G. Labah. *Algorithms for computer algebra.* Kluwer Academic Publishers, 1991.

[Hew77] Carl Hewitt. Viewing control structures aspatterns of message passing. *Artificial Inteligence*, 8, 1977.

[Hon93] Hoon Hong. Risc-clp(real): Logic programming with non-linear constraints over the reals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[Hon94] Hoon Hong. Confluency of cooperative constraint solver. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1994.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 1994.

[KG94] T. Khedro and M.R. Genesereth. Modeling multiagent cooperation as distributed constraint satisfaction problem solving. In A. Cohn, editor, *ECAI94: 11th European Conference on Artificial Intelligence*, pages 249–253. John Wiley & Sons, Ltd, 1994.

[LF94] Y. Labrou and T. Finin. A semantic approach for KQML – a general purpose communication language for software agents. In *Proc of CIKM'94*, 1994. to appear.

[Lho93] Olivier Lhomme. Consistency techniques for numeric CSPs. In *Proc. IJCAI93, Chambery, (France)*, pages 232–238, August 1993.

[Lho94] Olivier Lhomme. Consistency techniques for numeric CSPs. Thèse de doctorat, University of Nice Sophia Antipolis - CNRS, Route des Colles, B.P. 145, 06903 Sophia Antipolis Cedex, FRANCE, 1994.

[Moo66] R.E. Moore. *Interval Analysis*. Prentice Hall, 1966.

[MR94] P. Marti and M. Rueher. Solving Constraints Over The Reals — A Multi-Solver Approach. In *Proceedings of TAI'94 - 6th IEEE international conference on Tools with Artificial Intelligence*, pages 304–310. IEEE Computer Society Press, 1994.

[MRS94] U. Montanari, F. Rossi, and V. Saraswat. CC programs with both in- and non-determinism: A concurrent semantics. In Alan Borning, editor, *Proc. of PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, 1994.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, October 1979.

[PB94] G. Pesant and M. Boyer. QUAD-CLP(R): Adding the power of quadratic constraints. In Alan Borning, editor, *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle, May 1994.

[Rue93] Michel Rueher. A first exploration of prologIII's capabilities. *Software–Practice and Experience*, 23:177–200, 1993.

[Rue95] M. Rueher. *An Architecture for Cooperating Constraint Solvers on Reals*. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, pages 231–250. Springer Verlag, March 1995.

[Sar93] Vijay Anand Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[SKL90] Vijay A. Saraswat, Kenneth M. Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings North American Conference on Logic Programming*, 1990.

[Sun90] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Journal of Concurrency: Practice and Experience*, 2(4):315–339, December 1990.