Reliable Computing **3:** 325–333, 1997.
© 1997 *Kluwer Academic Publishers. Printed in the Netherlands.*

Concurrent Cooperating Solvers over Reals

MICHEL RUEHER
Université de Nice—Sophia Antipolis, I3S, Route des colles BP 145, 06903 Sophia Antipolis, France, e-mail: rueher@unice.fr

and

CHRISTINE SOLNON
Université Lyon I, LISI, Bat. 710–43, bd du 11 novembre 1918, 69622 Villeurbanne Cedex, France, e-mail: csolnon@bat710.univ-lyon1.fr

(Received: 29 November 1996; accepted: 28 February 1997)

Abstract. Systems combining an interval narrowing solver and a linear programming solver can tackle constraints over the reals that none of these solvers can handle on their own. In this paper we introduce a cooperating scheme where an interval narrowing solver and a linear programming solver work *concurrently*. Information exchanged by the solvers is therefore handled as soon as it becomes available. Moreover, to improve the pruning, the linear programming solver computes the actual range of values of each variable with respect to the subset of linear constraints. To validate the proposed architecture a prototype system—named CCC—has been developed. Several examples are given to illustrate the gain in speed and precision we can expect with CCC.

1. Introduction

Many industrial applications ranging from financial applications and resource allocation to thermal flow problems and electro-mechanical engineering problems involve solving arbitrary constraints over the reals. Such systems of constraints are usually mixed: they are formed of linear equalities and inequalities, non-linear equalities and often non-polynomial ones.

Interval arithmetic has been introduced in the constraint logic programming (CLP) framework [1], [2], [8], [11], [14], [21] because of its capabilities to narrow the domains of the variables for any system of constraints over the reals. We collectively call the local consistency algorithms used in these systems interval narrowing (IN) algorithms[⋆]. Chiu and Lee [4], [5] show that existing CLP languages based upon IN algorithms are deficient in handling linear systems of constraints over reals and therefore fail to solve general constraints systems over reals. For instance, IN fails to solve such trivial systems as $\{x + y = 2,\ x - y = 1\}$.

The motivation of cooperating approaches is to tackle such mixed systems of constraints over the reals by combining solvers based on different algorithms. Many systems combining a linear programming (LP) solver and an IN-solver have been proposed during the last years [3]–[5], [13], [16]. The purpose of this paper is to

[⋆] The overall scheme of the IN algorithm is given in Section 2.1.

introduce and to study a novel cooperating architecture based on the following features:

- a concurrent architecture;
- an active use of the LP-solver.

1.1. Concurrent Architecture

Cooperating solvers are usually based on the scheme suggested by Hong [9] which consists in repeatedly applying each sub-solver until no change occurs. The main drawback of such a cooperating process comes from its sequential nature: the new constraints inferred by one solver will only be considered by the other one when it has finished its resolution. As a consequence, new inferred constraints will often remain in the message queue of a solver while the latter is tediously trying to infer weaker constraints, and thus, the efficiency of the whole system may be jeopardized.

To tackle this kind of problems, we propose a cooperating scheme where an IN-solver and a LP-solver work *concurrently*. Each solver is composed of a local constraint store, a communication process and several solving processes. The communication process, which is in charge of handling the constraints sent by the other solver, works concurrently with the solving processes so that each newly received constraint is actually incorporated into the local store as soon as possible.

This concurrent architecture provides an elegant solution to some slow convergence phenomena of the classical IN-algorithm. The cycling problem occurs when a sequence $u_k = f(u_{k-1})$ slowly converges towards a fixed point u such that $u = f(u)$. If the linear sub-system allows the LP-solver to actually reduce the interval associated with some variables occurring in the cycle, concurrent cooperation makes it possible to cut down cycling. For instance, let us consider the following example:

$$\begin{array}{llllll} x_1 = 0.99 \times x_2 & (1) & x_4 = x_3^3 & (3) & & \\ x_3 = x_1 \times x_2 & (2) & x_2 = x_1 & (4) & & x_1, x_2, x_3, x_4 \in [0 \ldots 10000] \end{array}$$

In this example IN algorithm will run into an asymptotic convergence. Of course, a LP-solver will immediately derive from (1) and (4) the system $\{x_1 = 0,\ x_2 = 0\}$. Thanks to concurrency, this information will be considered by the IN-solver as soon as it is available and cycling will be cut down.

Conversely, the IN-solver may deduce relevant information concerning some variables a long time before it reaches a fixed point. In our concurrent architecture, these new bounds are actually sent to the LP-solver as soon as they are inferred (and not only when a fixed point has been reached).

1.2. Active Use of the LP-Solver

In most cooperating approaches, the LP-solver is used in a "passive" way: It only checks if the subset of linear constraints is consistent. Some LP-solvers also detect “fixed variables”, i.e., variables that can only take a single value. We propose in this paper to go one step further and to use the LP-solver in an active way for computing the actual range of values of each variable with respect to the sub-set of linear constraints. Let us consider for example the following set of constraints:

$$x_1 - x_2 \leq 2 \quad (1) \qquad -x_1 + x_3 \leq -1 \quad (3)$$
$$x_1 + x_2 \leq 1 \quad (2) \qquad -x_1 - x_3 \leq 0 \quad (4) \qquad x_1^2 \geq 4 \quad (5)$$

The subset of linear inequalities $\{1, 2, 3, 4\}$ is consistent but no variable is fixed. In such a case, a “passive” LP-solver, which only checks for consistency and detects fixed variables, will not allow to reduce the domains. However, the actual domain of variable x_1 with respect to the subset of linear inequalities is $[0.5, 1.5]$. Our “active” LP-solver will compute the bounds of this domain; bounds which are required to allow an IN-solver to detect the inconsistency of the whole system.

Hence, we propose to combine an IN-solver with an active LP-solver which actually computes the range of values of each variable with respect to the sub-set of linear constraints. The computed bounds are forwarded to the IN-solver which can use them to perform further narrowing.

1.3. Related Work

ICE [3] and PrologIV [16] have proposed a cooperation between an IN-solver and a LP-solver. In ICE the IN-solver signals each bound it has inferred for variables which are also included in some linear constraints, while the LP-solver sends to the IN-solver the values of shared variables as soon as they are fixed. The algorithms of PrologIV include Gauss and Simplex algorithms for handling equalities and linear inequalities over rationals, and an IN algorithm for approximating solutions of non-linear constraints over reals. Communication between these algorithms is based on the generation of a new constraint over the reals (resp. over rational numbers) for each fixed variable in the constraint system over rational numbers (resp. over the reals).

`CIAL` [4], [5] integrates in a CLP language two cooperating interval solvers: a linear equality solver based on an adaptation of the preconditioned interval Gauss-Seidel method, and an inequalities and non-linear equalities solver using IN algorithms and domain splitting techniques. The two solvers share common interval variables and are activated sequentially (in a round robin fashion) until a fixed-point is reached. Lhomme et al. [12] have proposed an algorithm for identifying and optimizing asymptotic convergence dynamically.

1.4. Outline of the Paper

Section 2 introduces the overall cooperating framework between an IN-solver and a LP-solver. It also shows how the IN-solver and the LP-solver work inside this framework. Implementation, experimentations and extensions issues are addressed in Section 3.

2. Cooperating Architecture

Our `CCC` system is composed of an IN-solver and a LP-solver which work concurrently. Each solver is defined by:

- a local constraint store;
- several solving processes;
- a communication process.

The resolution starts by sending the set of non-linear constraints to the IN-solver and the set of linear constraints to the LP-solver. The local constraint store of each solver is initialized and its solving processes are activated concurrently. Each solver forwards to the other one the new constraints it has inferred. Forwarded constraints are of the form $x_i \leq a$ or $x_i \geq a$ where x_i is a variable shared by both solvers and a a floating point number. This communication is based upon classical notions of the `cc` framework [19], [22], e.g., forwarded constraints have the form of *Tell* messages. Message passing between distributed solvers is based on the communication protocol introduced in [18].

The original point of this architecture is that the *communication process and the solving processes work concurrently*, i.e., the constraints sent out by a solver are actually incorporated within the constraint store of the other solver while its solving processes are still working.

2.1. The IN-Solver

CLP systems like BNR-Prolog [14], Newton [1], CLP(BNR) [2], Interlog [11], and Prolog IV [16] use a narrowing algorithm—adapted from the Waltz algorithm [23]—which computes an approximation of arc consistency [8]. The general scheme of this fixed-point algorithm is given in Figure 1. $\mathcal{C} = \{C_1, \ldots, C_m\}$ denotes a set of constraints over $\mathcal{X} = \{x_1, \ldots, x_n\}$, an arbitrary set of variables; $\mathcal{D} = \{D_1, \ldots, D_n\}$ denotes the associated set of domains and $Var(C_i)$ denotes the subset of variables of $\mathcal{X}$ which occur in C_i. Existing *CLP* systems over intervals mainly differ by the *narrowing* operator used in `IN-1`: systems like `CLP(BNR)` and `Interlog` use as narrowing operator an approximation of the projection functions[⋆] derived from the constraints while `Newton` uses as narrowing operator a variant of the Newton interval method.

`IN-1` is an incremental algorithm. However, when slow convergence phenomena occur, new bounds computed by the LP-solver may remain in the message queue for a very long time. As said before, this is a critical problem because some slow convergence phenomena can be removed with the information provided by the LP-solver.

[⋆] A projection function computes the set of possible values for a given variable of constraint C when all other variable domains are restricted to their solution set [11].

```
IN–1 (in $\mathcal{C}$, inout $\mathcal{D}$)
  $Q \leftarrow \{< x_i, C_j > \mid C_j \in \mathcal{C}$ and $x_i \in Var(C_j)\}$
  while $Q \neq \emptyset$
      $< x_i, C_j > \leftarrow$ POP($Q$)
      $\mathcal{D}' \leftarrow$ narrowing($\mathcal{D}, x_i, C_j$)
      if $\mathcal{D}' \neq \mathcal{D}$ then
          $\mathcal{D} \leftarrow \mathcal{D}'$
          $Q \leftarrow Q \cup \{< x_k, C_k > \mid C_k \neq C_j \wedge x_i \in Var(C_k)\}$
      endif
  endwhile
```

Figure 1. Scheme of the standard IN algorithm.

```
Repeat
    GetNewCst(C)
    if $Q \neq \emptyset$
        then enqueue($\{< x_i, C > \mid x_i \in Var(C)\}$)
        else resume IN-1($C, \mathcal{D}$)
    endif
endrepeat
```

Figure 2. Working Scheme of the Communication Process.

2.1.1. *Handling of a* `Tell` *Message*

The main idea is that a new constraint C carried by a `Tell` message must be taken into account before `IN-1` has reached a fixed-point. Indeed, as `IN-1` monotonically reduces the domains of the variables there is no reason to wait until a fixed-point has been computed for adding a new constraint to the constraint store. Thus, we propose here updating the queue Q while algorithm `IN-1` is working. The working scheme of the communication process is given in Figure 2. `GetnewCst(C)` reads a new constraint C from the message queue if available, otherwise it will wait until a new constraint arrives. To warrant the termination of the cooperating process, we assume that no new constraint introduces any new variable.

In this concurrent framework, new constraint C will be processed in the worst case after $O(m)$ iteration steps of `IN-1`; as a matter of fact, if procedure *enqueue* puts the new constraint C on the head of Q, it will be handled immediately.

2.1.2. *Sending out a* `Tell` *Message*

Deciding which constraints have to be forwarded is a problem similar to the one of handling constraints carried by `Tell` messages. If the interval solver works sequentially, the modified bounds are only forwarded when a fixed-point is reached, i.e., when no more reduction can be performed by IN alone. Thus, relevant information may be kept a long time in the local store of the interval solver. An alternative solution would consist in forwarding every new bound at each iteration step of `IN-1`. However, in this case, the LP-solver would overflow with information.

Thus, we propose to modify `IN-1` in order to be able to forward more concise and more relevant information. The main idea consists in performing concurrently all narrowing which may reduce some variable such than only the strongest reduction will be forwarded. Thus, instead of en-queuing tuples <variable, constraint> we will enqueue the variables whose domain may be reduced; the resulting algorithm is straightforward and its theoretical complexity is the same as the one of `IN-1`. The main advantage of this algorithm is that it forwards only the strongest reduction when several constraints can be used to reduce a bound of a given variable. Moreover, contrary to `IN-1`, it would be easy with this algorithm to evaluate in parallel all narrowing operators that work on a same variable.

2.2. The LP-Solver

The LP-solver has to compute the actual range of values of each variable with respect to the sub-set of linear constraints. It is composed of a local constraint store, a communication process and, for each variable x_i, two solving processes. These solving processes, called min(x_i) and max(x_i), respectively are in charge of computing the minimum and maximum values that x_i can take with respect to the sub-set of linear constraints.

Each solving process uses the simplex algorithm. New bounds sent by the IN-solver are repeatedly taken into account by packages, each time a solving process has finished its previous computation. The key point concerns the method for handling these (numerous) new bounds. Obviously, one cannot simply add to the simplex a new constraint, each time a new bound is received. Hence, we use *a simplex with explicit bounds* [6] so that the new bounds are taken into account by simply modifying the corresponding bounds in the simplex. In some cases, these modifications lead to a simplex which is still in feasible form, and no further computations are needed. However, if these modifications lead to a simplex in non feasible form, some iterations have to be performed. In this case, we perform *a dual optimization* [6] so that the new bound is computed in an incremental way: usually, the new optimal value is very close to the last computed one and very few dual iterations are needed to reach the new optimum.

3. Implementation and Discussion

A prototype of both a concurrent IN-solver and a concurrent LP-solver has been developed in `Oz`. `Oz` is a very high level programming language designed for symbolic computation [20]. `Oz` is based on logic variables and fair concurrency. The shared constraint store and the concurrent processes could thus been implemented in a straightforward way. For practical reasons, the IN-solver only forwards new

	Example 1	
$x_2 = 0.99x_1$	$x_1 = x_2$	$x_3 = x_1x_2$
$x_4 = x_2^2$		
	Example 2	
$x_1 - x_2 \leq 2$	$x_1 + x_2 \leq 1$	$x_1^2 \geq 4$
$-x_1 - x_3 \leq 0$	$-x_1 + x_3 \leq -1$	
	Example 3	
$u_1^2 + v_1^2 - 2 = 0$	$u_1 + v_1 - 2 = 0$	$u_1 - w_1 =< 0$
$w_1 - v_1 =< 0$	$x_1 + y_1 + z_1 + t_1 - w_1 = 0$	
$x_1 + y_1 + z_1 - t_1 - 2w_1 = 0$	$x_1 + y_1 - z_1 + t_1 - 3w_1 = 0$	
$x_1 - y_1 + z_1 + t_1 - 4w_1 = 0$	$y_1 - x_1 + z_1 + t_1 - s_1 = 0$	
$r1^2 + s_1 =< 0$	$u_2^2 + v_2^2 - 2w_1 = 0$	
$u_2 + v_2 - 2w_1 = 0$	$u_2 - w_2 =< 0$	$w_2 - v_2 =< 0$
$x_2 + y_2 + z_2 + t_2 - w_2 = 0$	$x_2 + y_2 + z_2 - t_2 - 2w_2 = 0$	
$x_2 + y_2 - z_2 + t_2 - 3w_2 = 0$	$x_2 - y_2 + z_2 + t_2 - 4w_2 = 0$	
$y_2 - x_2 + z_2 + t_2 - s_2 = 0$	$r2^2 + s_2 =< 0$	

Figure 3. Motivating examples.

bounds when 10% of the size of an interval has been removed or when a fixed point has been reached.

3.1. Experimental Results

To evaluate the effective contribution of our cooperating scheme—named `CCC`—we first compared it with the IN-solver working alone. `CCC` is always faster than the IN-solver: For all examples where cycling occurs, the improvement factor is more than 10. `CCC` also achieves stronger domain reduction than the IN-solver in numerous cases.

To better illustrate the advantages of `CCC`, we now compare it with `PrologIV` and `CIAL` on three small examples (cf. Figure 3). All systems are run on the same computer but time comparison is not really relevant since `PrologIV` is compiled whereas `CCC` and `CIAL` are interpreted.

In Example 1, the linear sub-system is strong enough to fix both x_1 and x_2 in any linear solver. Thus, both `PrologIV`, `CIAL` and `CCC` return the answers instantaneously.

In the Example 2, `PrologIV` cannot achieve any domain reduction while `CCC` detects the inconsistency. This is due to the fact that the linear sub-system is not strong enough to allow `PrologIV` to fix any variable. `CCC` computes the bounds of each variable and forwards them to the IN-solver which detects the inconsistency. CIAL computes a sharper approximation of the domain of values than IN algorithms and will therefore also detect the inconsistency.

In Example 3, PrologIV can only prune the domains of variables u_1 and u_2 while CCC computes a sharp approximation of the domain of values of all variables in a few seconds. To obtain this result, several exchanges of information between the solvers are required. It is worthwhile to notice that, in this case, CCC achieves a better pruning than solvers that use higher consistencies (like path consistency on bounds [11]). CIAL requires about 2 minutes for computing a very rough approximation of the solution. After domain splitting was invoked, CIAL did not return anything in 15 minutes or more.

3.2. Extensions

CCC is only an exploratory prototype and various enhancements would be required to compete with commercial systems. Among others, one can mention trivial formal simplifications (e.g., variable substitutions) and adding redundant linear constraints derived from non linear ones.

A critical point of CCC comes from the fact that the simplex algorithm is not correct with floating point numbers so that rounding errors occur rather frequently. In a practical way, to prevent some of the inconsistencies due to rounding errors, a small value (e.g., $1.\,10^{-10}$) is added (resp. subtracted) to the bounds forwarded by the LP-solver. Such an overestimation is also used in working with floating point numbers while using the preconditioned interval Gauss-Seidel method [5]. Moreover, the latter method cannot handle inequalities and its effective complexity is much higher than the one of the simplex. Another practical solution which seems promising consists in using a revised simplex method which usually has a better numerical stability and is also more efficient [17]. However, the only way for actually preventing all rounding errors would be to use rational numbers, instead of floating ones.

4. Conclusion

The key points of the framework introduced in this paper are *a tight cooperation between a LP-solver and an IN-solver*, and *the computation of the actual bounds over the linear sub-system*. The proposed concurrent architecture provides an elegant way to cut down part of the slow convergence phenomena of the classical IN algorithm. Moreover, exchanging information about the current values of the bounds prevents part of the early quiescence problems of IN and drastically improves the pruning. First experimental results on a prototype implementation indicate the relevance of the proposed framework.

References

1. Benhamou, F., Mc Allester, D., and Van Hentenryck, P.: CLP (Intervals) Revisited, in: *Proc. ILPS'94*, MIT Press, 1994.
2. Benhamou, F. and Older, W.: Applying Interval Arithmetic to Real, Integer and Boolean Constraints, *Journal of Logic Programming* (1994).
3. Beringer, H. and de Backer, B.: Combinatorial Problem Solving in Constraint Logic Programming with Cooperative Solvers, in: Beierle, C. and Plumer, L. (eds), *Logic Programming: Formal Methods and Practical Applications*, Elsevier Science Publishers, 1995.
4. Chiu, C. K. and Lee, J. H. M.: Towards Practical Interval Constraint Logic Programming, in: *Proc. ILPS'94*, MIT Press, 1994, pp. 109–123.
5. Chiu, C. K. and Lee, J. H. M.: *Efficient Linear Equality Solving in Constraint Logic Programming*, Research Report, Dep. of Computer Sciences, The Chinese University of Hong Hong, 1996.
6. Chvatal, V.: *Linear Programming*, W. H. Freeman and Co, 1983.
7. Colmerauer, A.: An Introduction to PROLOG-III, *Communications of the ACM* **33** (7) (1990), pp. 69–90.
8. Davis, E.: Constraint Propagation with Interval Labels, *Artificial Intelligence* **32** (1987), pp. 281–331.
9. Hong, H.: *Confluency of Cooperative Constraint Solvers*, Technical Report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1994.
10. Jaffar, J. and Maher, M.: Constraint Logic Programming: A Survey, *Journal of Logic Programming* **19** (20) (1994), pp. 503–581.
11. Lhomme, O.: Consistency Techniques for Numeric CSPs, in: *Proc. IJCAI93*, Chambery, France, 1993, pp. 232–238.
12. Lhomme, O., Gotlieb, A., Rueher, M., and Taillibert, P.: Boosting the Interval Narrowing Algorithm, in: *Proc. JICSLP'96*, MIT Press, 1996, pp. 378–392.
13. Marti, P. and Rueher, M.: A Distributed Cooperating Constraints Solving System, *Special Issue of IJAIT (International Journal on Artificial Intelligence Tools)* **4** (1–2) (1995), pp. 93–113.
14. Older, W. and Vellino, A.: Constraint Arithmetic on Real Intervals, in: Benhamou, F. and Colmerauer, A. (eds), *Constraint Logic Programming: Selected Research*, MIT Press, 1993.
15. Podelski, A. (ed.): *Constraint Programming: Basics and Trends, LNCS 910*, Springer Verlag, 1995, pp. 1–21 (Châtillon-sur-Seine Spring School, France, 1994).
16. *PrologIA. PrologIV—Constraint Inside (Handbook of PrologIV)*, Parc Technologique de Luminy—Case 919, 13288 Marseille cedex 09, France, 1996.
17. Refalo, P. and Van Hentenryck, P.: CLP($\mathcal{R}_{lin}$) Revisited, in: *Proc. JICSLP'96*, MIT Press, 1996, pp. 22–36.
18. Rueher, M.: *An Architecture for Cooperating Constraint Solvers on Reals*, in [15].
19. Saraswat, V. A.: *Concurrent Constraint Programming*, MIT Press, 1993.
20. Smolka, G.: *An Oz Primer*, DFKI Report, http://ps-www.dfki.uni-sb.de/oz/.
21. Van Hentenryck, P., Mc Allester, D., and Kapur, D.: Solving Polynomial Systems Using Branch and Prune Approach, *SIAM Journal*, to appear.
22. Van Hentenryck, P., Saraswat, V. A., and Deville, Y.: *Design, Implementation, and Evaluation of the Constraint Language* `cc(FD)`, in [15].
23. Waltz, D. L.: *Generating Semantic Descriptions from Drawings of Scenes with Shadows*, Tech. Report AI-TR-271, MIT Cambridge, 1972.