Satisfiability of Boolean formulas over linear constraints

Henri BERINGER, Bruno DE BACKER
IBM CEMAP
68/76 Quai de la Rapee
75592 PARIS CEDEX 12
{beringer,bdb}©vnet.ibm.com

Abstract

Testing the satisfiability of a Boolean formula over linear constraints is not a simple matter. Existing AI systems handle that kind of problems with a general proof method for their Boolean parts and a separate module for combining linear constraints. On the contrary, traditional operations research methods need the problem to be transformed, and solved with a Mixed Integer Linear Programming algorithm. Both approaches appear to be improvable if no early separation is introduced between the logical and numerical parts. In this case, combinatorial explosion can be dramatically reduced thanks to efficient looking-ahead techniques and learning methods.

Indeed, propagating bounds following the initial formula gives precious information. Besides, an especially tight linear relaxation can be driven from the formula, and allows a Simplex algorithm to make a good test for satisfiability. Finally, these two looking-ahead methods can be easily coupled for more efficiency and completed by local enumeration.

Moreover, discovering a good failure explanation is relatively easy in the proposed framework. By "learning" these explanations, it is possible to prune important redundant parts of the search tree.

area/subarea: Automated reasoning, Constraint satisfaction

1 Introduction

It is now widely accepted that general theorem provers or problem solvers cannot be efficiently implemented without relying on modules specialized in solving some well delimited problem classes [Lassez, 1991]. In particular, the most natural formalization of many concrete problems often relies on arithmetics over a continuous domain (as real or rational numbers) which cannot be handled directly by general theorem-proving algorithms such as resolution. In such a case, it is often proposed to still use a general theorem-proving algorithm which handles arithmetic constraints as uninterpreted propositions, and to control this proof by calling a constraint solving module able to check the compatibility between arithmetic constraints. This is, for example, what is done with the CLP(R) language.

Unfortunately, such a strong decomposition limits drastically the performance of the resulting algorithm. This is the reason why we propose here to solve directly problems combining logical and arithmetic constraints. More precisely, this paper deals with the satisfiability problem for formulas built with Boolean operators, Boolean variables and linear constraints. Note that this is a language powerful enough to model many physical systems by piecewise linear approximation, for example electronic circuits or mechanical systems.

After describing precisely the problem to solve, we will present the main components of the proposed algorithm. This algorithm, based on implicit enumeration, is complete. Moreover, it succeeds in reducing combinatorial explosion in two ways. Firstly, it "looks ahead" efficiently by both testing the satisfiability of a "tight" linear relaxation of the overall problem, and by inferring the value of some decision variables early thanks to bounds propagation. Secondly, it is able to learn at low cost from failure analysis, and then avoids many redundant computations.

2 The Problem

The problem to be solved is the satisfiability of formulas having the following syntax:

$$\begin{array}{rcl}
LForm & ::= & LForm \wedge LForm \\
LForm & ::= & LForm \vee LForm \\
LForm & ::= & LForm \Rightarrow LForm \mid \neg LForm \\
LForm & ::= & Proposition \mid LCst \\
LCst & ::= & LExpr = LExpr \mid LExpr \leq LExpr \mid \\
 & & LExpr \geq LExpr \\
LExpr & ::= & Real_Num \mid Real_Num * Real_Var \\
LExpr & ::= & LExpr + LExpr \mid LExpr - LExpr
\end{array}$$

This language allows many problems to be described. In particular, it can be used to model physical systems by piecewise linear approximation. Thus, in the domain of analog electronic circuit diagnosis the following formulas about a transistor's good behavior have been successfully used in [Dague *et* al., 1991]:

$$
\begin{aligned}
& correct(transistor_t1) \Rightarrow \\
& \quad I_t1_c \geq -1.E-4 \\
\wedge & \quad (V_t1_b_e \geq 0.6 \wedge V_t1_c_e \geq 0.3 \\
& \quad\quad \Rightarrow I_t1_c \geq 50 * I_t1_b \wedge I_t1_c \leq 100 * I_t1_b) \\
\wedge & \quad (V_t1_b_e \leq 0.4 \Rightarrow off(transitor_t1))
\end{aligned}
$$

where $correct(transitor_t1)$ and $off(transistor_t1)$ are propositions meaning respectively that the transistor $t1$ is working correctly and that the same $t1$ is in blocking mode. I_t1_c and I_t1_b represent currents, and $V_t1_b_e$ and $V_t1_c_e$ are voltage drops.

The overall satisfiability problem can be encountered in several applications as design, verification or diagnosis. In each of these application domains, it is possible that the proposed algorithm should be modified. In any case the basic technology presented here remains valid.

3 A normal form

In order to simplify the exposal, we suppose the problem formulation has been normalized according to the following rewriting rules:

$$
\begin{aligned}
X \Rightarrow Y & \rightarrow \neg X \vee Y \\
\neg(X \vee Y) & \rightarrow \neg(X) \wedge \neg(Y) \\
\neg(X \wedge Y) & \rightarrow \neg(X) \vee \neg(Y) \\
LExpr1 = LExpr2 & \rightarrow LExpr1 \leq LExpr2 \\
& \quad\ \wedge LExpr1 \geq LExpr2 \\
LExpr1 \geq LExpr2) & \rightarrow -LExpr1 \leq -LExpr2 \\
\neg(LExpr1 \leq LExpr2) & \rightarrow LExpr2 + \epsilon \leq LExpr1
\end{aligned}
$$

where ϵ is a small enough number dependent of the application (if necessary, this ϵ may be handled symbolically). The result of this rewriting operation is then a formula built with only "and" and "or" applied to propositions (possibly negated) and to linear constraints which may be written $a.x \leq b$ where x is a column vector of variables, a is a line vector of constants, b is a constant, and "." denotes the internal product. Such a formula follows the syntax:

$$
\begin{aligned}
Norm_Form & ::= Disj_C \wedge ... \wedge Disj_C \mid Disj_C \\
Disj_C & ::= Norm_Form \vee ... \vee Norm_Form \mid \\
& \quad\ a.x \leq b \mid Propostn \mid \neg Propostn
\end{aligned}
$$

The example of section 2 is thus rewritten as :

$$
\begin{aligned}
& \neg correct(transistor_t1) \\
& \vee (\quad -I_t1_c \leq 1.E-4 \\
& \quad \wedge \quad (V_t1_b_e \leq 0.6-\epsilon \vee V_t1_c_e \leq 0.3-\epsilon \\
& \quad \vee \quad (-I_t1_c + 50 * I_t1_b \leq 0 \\
& \quad\quad \wedge I_t1_c - 100 * I_t1_b \leq 0) \\
& \quad \wedge \quad (-V_t1_b_e \leq -0.4 - \epsilon \vee off(transitor_t1))
\end{aligned}
$$

Below, in order to further simplify the presentation, a positive proposition $Prop$ will be replaced by the linear constraint $Prop = 1$ and $\neg Prop$ by $Prop = 0$ where $Prop$ is considered as a real variable about which the constraint $Prop = 0 \vee Prop = 1$ is added to the problem. In the practical implementation such a transformation is not performed.

4 A basic and complete solving algorithm: enumeration

A basic but always useful algorithm for solving combinatorial problems is implicit enumeration. Its principle is to explore systematically the search space by successively dividing it, i.e., choosing alternatives, until the problem becomes simple enough to be directly solved.

Thus, it is possible to solve the above problem by a so called "depth-first backtrack search" (see for example [Dechter, 198990]). After making all the possible choices (elements in disjunctions), a simple conjunction of linear constraints is obtained. Such a system is then easily tested for solvability by an efficient algorithm such as the Simplex.

As the Simplex is a complete algorithm, if all the possible choice combinations are tried, the overall algorithm is complete. However, it may spend a time proportional to the exponential of the number of disjunctions. Fortunately, this can be dramatically improved by looking-ahead and learning techniques as presented below.

5 Looking-ahead techniques

During the search, each time a subproblem is generated, even if the resulting formula is still not directly solvable by a Simplex because of remaining disjunctions, it is possible to analyze it and reduce a priori the remaining search space by removing some alternatives. Sometimes, the analysis will detect, without combinatorial search, that the subproblem is unsatisfiable. Two complementary analysis methods may be used: constraint propagation and linear relaxation. Especially interesting methods of these two kinds are presented below.

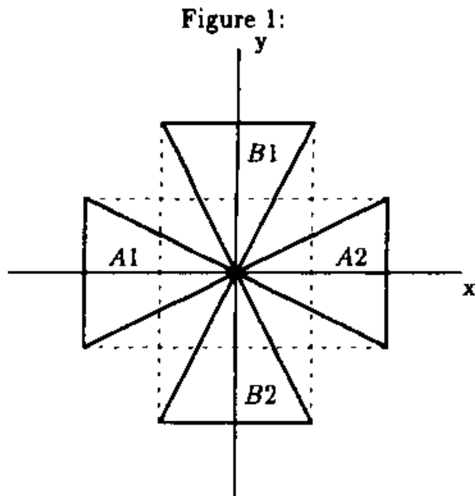
5.1 Bound propagation

Bound propagation based on linear constraints can easily be implemented. Indeed, given a single linear constraint and some initial bounds on its variables, it is quite simple to compute how the bounds of each variables may be changed: the constraint may be rewritten as $x \leq Expr$ or $x > Expr$ for any of its variables x. Given a conjunction of linear constraints, bounds may be propagated by using each constraint after the other until stability is reached (or some stopping criterion is verified). It is to be noted that this process is fairly incomplete. As an example, with the constraint set: $x = y$ and $x = -y$ and x and y in $[-1,1]$, this method fails to infer any stricter bound, though x and y are necessarily 0.

In order to apply bound propagation to any formula, it suffices to work bottom up in the formula tree computing bounds which may be inferred from each terminal conjunctions of linear constraints, then grouping the results for the disjunctions by selecting the less tightening bounds and so on until new bounds are computed for the root, i.e., for the whole problem. For example, if the formula $(A1 \vee A2) \wedge (B1 \vee B2)$ is considered where A_i and Bi are the triangles shown in figure 5.1, the first step of bound propagation yields the bounds represented by the box enclosed in the dashed lines. This example demonstrates that bound propagation can last forever:

its solution is (0,0) and at each step the bounds are only reduced by a factor of 2.

Figure 1:



In order to draw the best of such a method, it is important to propagate any new bound inferred at one level of the formula tree in its subtrees (i.e., to propagate bounds under hypotheses in the part of the formula which holds under the same hypotheses). This may be done with the algorithm described below. The recursive function propagate.bounds takes as arguments a set of initial Bounds for the variables, and a Formula. It returns either a set of bounds implied by Formula, or the information Unsolvable if the formula is discovered incompatible with the initial Bounds.

```
propagate_bounds(Bounds, Formula)

  if Formula is a single linear constraint
  for each variable x in this constraint update its bound
  in Bounds as described above and return the result
  (which may be Unsolvable)
  if Formula is a conjunction Formula1 and Formula2
  repeat
     Old_Bounds : = Bounds
     Bounds := propagate.bounds(Bounds, Formula\)
     Bounds := propagate.bounds( Bounds, Formula2)
  until Bounds = Unsolvable or Old-Bounds = Bounds
     (or after n iterations)
  return Bounds
  if Formula is a disjunction Formula\ V Formula2
  Bounds] = propagate_bounds{Bounds, Formula\)
  Bounds2 — propagate' .bound s( Bounds, Formulay)
  if (Bounds1 = Unsolvable) return Bounds2
  if (Bounds2 — Unsolvable) return Bounds1
  else return Bounds] U Bounds2
  (union of each intervals)
```

The propagation is started by calling this function with Bounds containing no restriction and Formula being the whole formula to be tested for satisfiability.

The practical implementation of this algorithm is in fact more incremental. At each disjunctive node of the tree, the Bounds\ and $Bounds_2$ sets are cached. Then, whenever propagation of Bounds is requested for this node, only the new information contained in Bounds compared to $Bounds_1$ (or $Bounds_2$) is propagated through $Formula_1$ (respectively $Formula_2$). Note that, at the end of the algorithm, $Bounds_1$ (respectively $Bounds_2$) may be regarded as the bounds which hold under the hypothesis that the first (respectively second) alternative of the disjunction have been chosen (and therefore that the whole path leading to this disjunction has been selected).

Despite its incompleteness, bound propagation updated at each level of the search tree often reduces the combinatorial explosion in an important manner by discovering soon that some alternatives may be suppressed.

5.2 Linear relaxation

Solving a simple conjunction of linear constraints is an easy matter thanks to algorithms such as the Simplex. Even if a problem of the class studied here contains disjunctions, it is possible to derive from it a new problem containing only conjunctions and whose solution set includes every solution to the initial problem. This is called a linear relaxation. Testing the solvability of a relaxation already gives important information: if the relaxation is unsolvable, then so is the initial problem.

Obviously, a tight relaxation, i.e., one whose solution set is as small as possible, is of great interest. In geometrical terms, a set expressed using disjunctions of linear constraints is not convex, whereas one expressed using only conjunctions is convex. The idea is then to use the convex hull of a disjunction for its relaxation. The linear constraints defining this convex hull can be computed symbolically, by using a variable elimination algorithm [Lassez and Lassez, 1991]. This is however to be avoided, because the number of constraints defining the convex hull may grow exponentially with respect to the number of constraints appearing in the disjunction.

Nevertheless, there exists an interesting construction adapted from Balas [Balas, 1985] which allows the convex hull of a disjunction of linear constraint sets to be represented compactly by only adding new variables. It is presented here for a disjunction with two elements, but the extension to more elements is straightforward.

Let D be the set defined by $\{y \mid A_1.y \leq b_1 \vee A_2.y \leq b_2\}$ (here A_i are matrices and b_i are column vectors). Let x be an element of $conv(D)$, the convex hull of D. Then there exist reals y_1 and y_2, and positive reals δ_1 and δ_2 such that $x = \delta_1 y_1 + \delta_2 y_2$, $\delta_1 + \delta_2 = 1$ with $A_1 y_1 \leq b_1$ and $A_2 y_2 \leq b_2$. Let $x_1 = \delta_1 y_1$ and $x_2 = \delta_2 y_2$. This yields the relaxation $rel(D)$:

$$\begin{aligned} x &= x_1 + x_2 \\ 1 &= \delta_1 + \delta_2 \\ A_1 x_1 &\leq b_1 \delta_1 \\ A_2 x_2 &\leq b_2 \delta_2 \end{aligned}$$

In the definition of rel(D), the positivity constraints on the 6_t can actually be removed. This is one difference between Balas' result and the following theorem. Moreover, we have demonstrated it even for unbounded polyedron.

Theorem 1 $x \in conv(D) \Leftrightarrow x \in rel(D)$

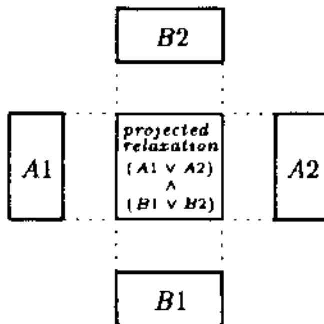
Proof 1

- ($\Rightarrow$) *By the above construction.*
- ($\Leftarrow$) *Let $c.x \leq d$ be a constraint implied by $conv(D)$. Then by Fourier's theorem there exist vectors λ_1 and λ_2 such that $\lambda_i A_i = c$ and $\lambda_i b_i \leq d$ for $i \in \{1,2\}$. This, together with $A_i x_i \leq b_i \delta_i$, gives $c.x_i \leq d\delta_i$, for $i \in \{1,2\}$. By summing these two constraints, we get $c.(x_1 + x_2) \leq d(\delta_1 + \delta_2)$, As $x = x_1 + x_2$ and $\delta_1 + \delta_2 = 1$, this means that any point x of $rel(D)$ verifies $c.x \leq d$.*

Now, any problem in normal form can be relaxed into a conjunction of constraints, using Balas' relaxation to remove each disjunction. Please note that, because of the conjunctions connecting different disjunctions with each other, it is not the convex hull of the solution of the global problem which is obtained, but instead a less tight constraint set.

The example in figure 5.2 illustrates how from the inconsistent formula $(A1 \vee A2) \wedge (B1 \vee B2)$, the space of possible solutions is well reduced by the relaxation without succeeding in discovering the inconsistency. Note that on this example bound propagation alone succeeds in proving inconsistency. However if the drawing is rotated by 45°, bound propagation does not find anything, while Balas' relaxation will be as precise as without rotation.

Figure 2:



This linear relaxation is only one possible among many. It is however the only known one which is so tight without needing any complex computation. Relaxations for a disjunction of constraints which are currently used in Mixed Integer Programming (see for example [Williams, 1988]) introduce less new variables in the formulation. However, they do not succeed in approaching the convex hull of the disjunctions. Balas' relaxation used here certainly adds many new variables, but because the average complexity of the Simplex is only proportional to the logarithm of the number of variables, this is not a real problem. Moreover, as the added constraints are only equalities, they can be easily erased by substitution. Therefore, Balas' relaxation is both computationally convenient and very precise.

5.3 Coupling bound propagation and relaxation

Balas' relaxation is perfect (i.e., represents the convex hull of the solutions) only when applied to one disjunction of constraint sets. However, it is in theory possible to build a convergent series of relaxations by:

- computing explicitly the projection on the initial x variables of the first Balas' relaxation (that suppose each variable x_i and δ_i are removed by an algorithm such as the one in [Lassez and Lassez, 1991]).
- adding the result to each leaf of the formula tree
- and iterating, i.e., recompute a Balas' relaxation...

Balas has demonstrated in [Balas, 1985] that such a series converges (in possibly infinite time) toward the convex hull of the solutions of the initial formula. This is completely impractical since just computing the projection may give an exponentially big result, which is to be duplicated and re-projected. Even so, this theoretical result is useful in showing the interest of "re-injecting" in the leaf of the tree the information about the global solution space. The only manageable information for this purpose is the bounds on the variables. Now, let us consider the bounds computed by the propagation for one leaf of the formula tree. By definition, these bounds can be added as explicit constraints to the corresponding formula leaf without removing any potential solution. Then, if Balas' relaxation is consequently updated, it may become much tighter.

Finally, relaxation can be used to perform a "local enumeration". This means verifying that each δ_i may be fixed to 1. If some δ_i cannot be set to 1 then they are necessarily zero and the corresponding alternative should be suppressed. After such suppressions it is worth to restart the looking ahead phase at its beginning, i.e., bounds propagation.

6 Learning

By learning, we mean inferring general Boolean constraints over hypotheses, in order to reduce a priori unexplored paths of the search tree. Such constraints can be detected using two different methods: incompatibilities from bounds, and failure analysis. In both cases, these constraints are of the form $\neg(\wedge_i \delta_i)$. During the constraint solving process, many of them can be inferred. They are stored in a symbolic Boolean constraint solver, or better in an ATMS [de Kleer, 1986]. Such an algorithm is able to detect early that hypotheses are fixed (in the case of an ATMS, one-element nogoods), and manages efficiently the constraints in store. Therefore, after many such constraints have been inferred, the possibilities for the hypotheses δ_i can be reduced drastically. For example, if δ_1 and δ_2 are known to be incompatible, if one of them is fixed to 1, then the choice corresponding to the other one will automatically be avoided.

6.1 Learning from bounds

Each bound computed by the bound propagation algorithm can be labeled by the set of hypotheses yielding to it. When bounds for the same variable are incompatible

with two different sets of hypotheses, for example, $x \leq 3$ for $\delta_1 \wedge \delta_2 \wedge \delta_3$ and $x \geq 4$ for δ_4, the conjunction of all the hypotheses is false. If an extended ATMS similar to the one described in [Dague *et al.*, 1991] is used, the bounds $x \leq 3$ and $y \geq 4$ are recorded along with their justification. A new nogood $\{\delta_1, ..., \delta_4\}$ is then created, and the ATMS propagates this information, possibly discovering new nogoods.

6.2 Failure analysis

Relaxing a problem yields a set *S* of linear constraints over continuous variables and δ variables. If it is unsolvable, it is interesting to know why. This can easily be analyzed as being due to a subset of constraints in *S* which are incompatible together. This subset will be called a conflict. As each constraint is related to one hypothesis δ_i, a conflict may be easily translated in terms of hypotheses.

Let us first explain how a conflict in terms of constraints may be built. The solvability of the set *S* $\{A.x \leq b\}$, where *A* is a matrix, can be tested by computing minx_0 on $Ax + Is - 1.x_0 = b$, where I is the identity matrix, $s = (s_1, ...s_n)$ is a vector of slack variables, x_0 is a positive variable, and 1 is a column vector whose coordinates are all equal to 1. *S* is solvable if and only if minx_0 = 0.

When *S* is unsolvable, the Simplex reaches a positive minimum value for *X0* with some variables in the basis (i.e., being not necessaraly stuck to 0). Constraints whose slack variables are not in this basis are the limits on which the minimization stumbled. They form a minimal conflict of *S.* As the reader can see, the detection of this conflict is done a posteriori, without any modification of the core constraint satisfaction algorithm, i.e., at no cost. For a detailed presentation of this result see [De Backer and Beringer, 1991].

Once a cause of inconsistency has been found in terms of constraints, it can be translated in terms of the corresponding hypotheses. It can be added as a nogood in the ATMS, so that it can be used to further reduce the possibilities on the hypotheses. Please note that the minimality of a conflict in terms of constraints does not guarantee at all that the corresponding conflict in terms of hypotheses is minimal. However, it is usually not far from being minimal.

7 Comparison with other approaches

Many AI systems just use some kind of bound propagation to draw information from any kind of arithmetic constraints. Depending on the application, the incompleteness of such a method is more or less disturbing. For example, in a diagnosis application such as [Dague *et* al., 1991], it may forbid to detect some failure cause. In any case, the hereabove tools may be fruitfully added to these systems when more completeness is necessary.

7.1 Mixed Integer Linear Programming

There exist many commercial packages able to solve Mixed Integer Linear Programming (MILP) problems, i.e., problems described by a simple conjunction of linear constraints in which some variables are forced to take integer values. The MILP formalism is powerful. In particular, the satisfiability of Boolean formulas over linear constraints may be transformed into a MILP problem.

Good MILP packages solve such problems by an implicit enumeration method controlled by a Simplex and bound propagation. So even without any learning, they normally perform not too badly. However, when applied to problem naturally expressed in the disjunctive form studied here, they have an important limitation: they are not able to deal directly with the disjunctive formulation. The user is then forced to reformulate completely the problem in MILP This means that the user is in charge of generating the relaxation used by the algorithm.

Now, though the relaxation proposed here is the most precise one without being computationally expensive, it cannot be used as such with MILP packages. Indeed, even after adding the constraints that the δ_i variables are equal to either 0 or 1, this relaxation still has solutions which do not correspond to solution of the initial formula. To obtain this essential property, many extra artificial constraints have to be added to enforce x_i to be zero when δ_i is zero. This is done by constraining each variable x, with two artificial bounds $x_i \leq M\delta_i$ and $x_i \geq -M\delta_i$ where *M* is a large enough positive number. The resulting formulation has so many constraints that the Simplex may perform poorly on it. This is perhaps the reason why, despite its qualities, Balas relaxation has been hardly ever used.

Moreover, whatever MILP reformulation is chosen by the user, it hides the initial structure of the problem in such a way that bound propagation applied to this reformulation makes much less inferences than when it is applied directly to the initial formulation as proposed here.

Finally, no MILP package seems to use a learning technique similar to the one proposed here.

As a result, on difficult problems naturally expressed as Boolean formulas over linear constraints, our algorithm should perform better than any MILP package running on any MILP reformulation.

7.2 Constraint Logic Programming

Languages as CLP(R) [Heintze *et ai,* 1991] which extend the Prolog language with linear constraints may be used to express the satisfiability problem considered here, and even to solve it. However, used as such, the resolution algorithm handles very poorly the disjunctions: they are not considered in any way before branching on them.

But note that, when used to solve combinatorial problems, CLP languages are more programming languages than ready-to-use solving algorithms. As a matter of fact, the authors are currently testing and implementing the methods presented here using CLAIRE, a prototypal constraint extension of IBM Prolog.

8 Conclusion

Solving Boolean formulas containing linear constraints is a difficult combinatorial problem. As for any combinatorial problem, there does not exist an "ideal" algorithm

which would be better than the others on every problem. Success in combinatorial problem solving lies in the adequacy of the solving strategy to the restricted problem set to be solved. In particular, the user must have a way to specify heuristics able to guide the search. Indeed, success of Constraint Logic Programming over finite domains [Hentenryck, 1989] may be explained by the ability to combine easily efficient algorithm thanks to a high level programming language.

The present work intended to provide basic procedures useful to design efficient algorithms while taking adavantage of the problem specificities. The different procedures proposed here are not only separately efficient, but are also perfectly complementary. Thus, learning by failure analysis often gets an information which is not easily found by a priori analysis; Balas' relaxation is locally perfect but does not integrate long distance influences; and, on the contrary, bound propagation is a rough approximation which takes into account the links in the overall problem.

The very first experiments with the prototype written in CLAIRE have shown good results. Further work will include the benchmarking of the method on problems of good size.

References

[Balas, 1985] E. Balas. Disjunctive programming and a hierarchy of relaxation for discrete optimization problems. SIAM J. Alg. Disc. Meth., 6(3), July 1985.

[Dague et a/., 1991] P. Dague, O. Jehl, and P. Taillibert. An interval propagation engine and conflict recognition engine for diagnosing continuous dynamic systems. In Workshop on model-based diagnosis. Springer-Verlag, Vienna, 1991.

[De Backer and Beringer, 1991] B. De Backer and H. Beringer. Intelligent backtracking for CLP languages, an application to CLP(R). In International Logic Programming Symposium, San Diego, 1991.

[de Kleer, 1986] J. de Kleer. An assumption-based TMS. Artificial Intelligence, 28, 1986.

[Dechter, 198990] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decompostion. Artificial Intelligence, 41, 1989/90.

[Heintze et al., 1991] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) programmer's manual. Technical report, IBM TJ. Watson Research Center, Yorktown Heights, November 1991.

[Hentenryck, 1989] P. Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, 1989.

[Lassez and Lassez, 1991] C. Lassez and J-L. Lassez. Quantifier elimination for conjunctions of linear constraint via a convex hull algorithm. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, 1991.

[Lassez, 1991] J-L. Lassez. From LP to LP: Programming with constraints. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, 1991.

[Williams, 1988] H.P. Williams. Model Building in Mathematical Programming. Wiley, 1988.