Finding Conflict Sets and Backtrack Points in CLP($\Re$)

Jennifer Burg
Wake Forest University
Winston-Salem, NC 27109
burg@mthcsc.wfu.edu

Sheau-Dong Lang
Charles E. Hughes
University of Central Florida
Orlando, FL 32816
lang@cs.ucf.edu
ceh@cs.ucf.edu

Abstract

This paper presents a method for intelligent backtracking in CLP($\Re$). Our method integrates a depth-first intelligent backtracking algorithm developed for logic programming with an original constraint satisfaction algorithm which naturally generates sets of conflicting constraints. We prove that if CLP($\Re$) is assumed to cover strictly the domain of real numbers, then the constraint satisfaction algorithm provides minimal conflict sets to be used as a basis for intelligent backtracking. We then extend the backtracking method to cover a two-sorted domain, where variables can be bound to either structured terms or real numbers. We discuss a practical implementation of the algorithm using a generator-consumer approach to the recording of variable bindings, and we give an example of a CLP($\Re$) program which benefits significantly from intelligent backtracking.

1 Introduction

CLP($\Re$) is a constraint logic programming language in which constraints can be expressed in the domain of real numbers. Even when these constraints are restricted to linear equations and inequalities[1], this language has proven to be expressive enough for practical applications such as scheduling problems, options trading, critical path analysis, resolution of temporal constraints, and AI-type puzzles [**?**, **?**]. However, the range of problems which are solved efficiently by CLP($\Re$) is limited by the possible explosion of the search space as well as the complexity of the constraint satisfaction algorithm. A number of approaches to speeding up execution of CLP($\Re$) programs have been

[1]with, optionally, a mechanism for "delaying" non-linear constraints until sufficient variables are bound to make them linear

explored, including the fine-tuning of simplex-based constraint satisfaction algorithms, compilation [?], parallelism [?], and intelligent backtracking, the subject of this paper.

Intelligent backtracking seems a particularly appropriate strategy for CLP($\Re$), since the real-number domain lends itself quite naturally to the identification of conflict sets. We can illustrate this point with the CLP($\Re$) program in Figure 1, which solves the following cryptarithmetic puzzle:

$$\begin{array}{ccc} & B & B \\ + & B & A \\ \hline E & D & C \end{array}$$

When node 4 is exited, A is 9 and B is 8. Upon exit from node $8'$, C is redundantly given the value 7. At node 15, we fail to find a consistent value for D. Clearly, this failure was inevitable as soon as A was given the value 9 and B the value 8, and trying new values for C is wasted effort. A useful intelligent backtracking algorithm should recognize this and backtrack directly to node 4.

The backtracking method which we propose for CLP($\Re$) prunes obvious failure paths from the search tree by identifying conflict sets during the constraint satisfiability check. The first time a failure occurs at node 17, the relevant constraints in the tableau are (assuming the unification of variables) (1) $A \geq 0$, (2) $B > 0$, (3) $C \geq 0$, (4) $D \geq 0$, (5) $A \leq 9$, (6) $B \leq 9$, (7) $C \leq 9$, (8) $D \leq 9$, (9) $A + B = 10 * CA + C$, (10) $B + B + CA = 10 * E + D$, (11) $E = 1$, (12) $CA = 1$, (13) $A = 9$, (14) $B = 8$, (15) $A - B > 0$, (16) $C = 7$, (17) $B - C > 0$, (18) $A - C > 0$, (19) $D = 9$, (20) $D < A$. A failure is signaled when D is given the value 9 but at node 16 the constraints $D < A$ and $D > A$ both fail. We will show that our constraint satisfaction algorithm can easily identify the source of the conflict (equations 20, 14, 13, 12, 11, 10, and 9), and as execution retreats from node 15 this information sends us directly back to the last node where one of the guilty equations was introduced, node 4.

The compatibility between CLP($\Re$) and intelligent backtracking has been recognized by DeBacker and Beringer [?], Hogger and Kotzamanidis [?], and Burg, Lang, and Hughes [?, ?]. The roots of the research can be traced to Bruynooghe and Pereira [?] and Cox [?], who developed strategies for intelligent backtracking in logic programming. More recent work includes generator-consumer approaches [?] and the DIB (depth-first intelligent backtracking) algorithm of Codognet, Codognet, Filé, and Sola [?, ?]. DeBacker and Beringer give a recursive formulation of the DIB algorithm in [?], pointing out that it is applicable to any constraint logic programming language in which conflict sets can be identified upon failure. However, they stop short of integrating the algorithm into CLP($\Re$) since they do not allow for the possibility of structured terms in the domain. With such a restriction, the CLP($\Re$) they assume is not the language originally conceived by Jaffar et al., who describe CLP($\Re$) as based upon a two-sorted domain of real numbers and structured terms [?]. The two-sorted domain yields a more interesting, expressive language, and it is worthwhile to extend intelligent backtracking to this context. Our contribution to this research area is to integrate intelligent backtracking into the two-sorted domain of CLP($\Re$), basing our method on an original constraint satisfaction algorithm and an accompanying proof that minimal conflict sets are generated directly.

2 Depth-First Intelligent Backtracking

We begin by sketching the DIB algorithm which forms the framework of our backtracking method. Consider the example in Figure 2. At each failure node i, we identify a set containing conflicting constraints associated with the failure. Call this set the *conflict set* for node i. The *source* of a constraint is defined as the node to which execution must return in order to remove the constraint from the tableau. The source of a goal literal is defined similarly. The *backtrack set* for failure node i is defined as the union of the source numbers for all the constraints in i's conflict set. The backtrack set tells us the nodes at which new execution paths ought to be tried. The idea is that if node i receives a failure message from its child node, but i is not in the accompanying backtrack set, then constraints introduced at i had nothing to do with the conflict, and trying another branch from there is useless.

Say that when we attempt to execute goal t at node 5, we fail on both branches. When the conflict is detected at node 6, the backtrack set $\{2, 5\}$ is passed up and stored at node 5. Then, node 5 tries another branch to execute goal t. Again, t fails, and the backtrack set $\{1, 5\}$ is returned to node 5, where it is unioned with $\{2, 5\}$. At this point, node 5 has no more paths to try, so 5 is deleted from the collected set. Note that before this backtrack set $\{1, 2\}$ can be sent up the tree, the source of node 5's leftmost goal must be included in it, yielding $\{1, 2, 4\}$. This is because it is goal t which failed, and when we go back to node 4 we remove t from execution. If 4 were not in the backtrack set returned to node 4, node 4 would not try another branch and a solution could be missed. In our example, re-execution of goal s at node $5'$ returns the backtrack set $\{2, 4\}$ to node 4. Since node 4's leftmost goal, s, was in execution from the root of the search tree, there is nothing to add to this backtrack set, and node 3 receives the set $\{1, 2\}$. At this point, node 3 knows that it is useless to try another branch, and it immediately fails, passing the backtrack set $\{1, 2\}$ up to node 2.

3 The Constraint Satisfaction Algorithm

The implementation of a CLP($\Re$) interpreter can be divided into two modules: an inference engine, which performs resolution; and a solver, which checks the satisfiability of linear constraints in the real number domain. To describe this algorithm, we use R and M to denote the tableau of constraints.[2] R represents the constraints as they first appeared to the

[2]For convenience, we speak interchangeably of the tableau as an ordered list of rows, a set of rows, or a matrix. M_i denotes the i^{th} row in the tableau, while $M_{i,j}$ denotes the coefficient of the j^{th} variable in the i^{th} row.

solver, in their canonical input form. Before being given to the solver, inequalities are transformed to equations with slack variables, where each slack variable is constrained to take on only non-negative values. M is the "working copy" of the tableau, upon which Gaussian elimination and the simplex method are performed in order to maintain M in a solved form.

We call our constraint satisfaction algorithm *checksat*. Beginning with a tableau M of $m-1$ constraints which, taken together, are known to be satisfiable, *checksat* is given a new constraint to process, identical copies being placed in rows R_m and M_m. Each row M_i in the satisfiable tableau has a variable that is implicitly being "solved for" in the row. We call this variable the *basic variable* in the row. If the basic variable is a program variable (unconstrained), then it is the first variable in the row.[3] If the basic variable is a slack variable, then it does not appear in any other row which has only slack variables (*all-slack rows*). The constant in each row in the tableau is maintained as non-negative.

The tableau is maintained in this solved form in the following manner. First, basic unconstrained variables are substituted out of M_m until the first non-basic unconstrained variable is encountered, if one exists. Then there are four cases to consider. (1) If the equation has reduced to $0 = 0$, it is redundant and M_m and R_m are discarded. (2) Else, if it reduces to $a = b$ where a and b are constants and $a \neq b$, it is in conflict with the rest of the tableau. (3) Else, if a non-basic unconstrained variable has been encountered, it is made basic. (4) Else, if only slack variables remain, basic slack variables are substituted out of M_m, the all-slack rows are isolated, and we perform the first phase of the two-phase simplex method on them to determine if a slack variable can be made basic. We know that a slack variable cannot be made basic if we reach a state where the coefficients for all the slack variables in the new row are non-positive while the constant is positive. In this case, the tableau is unsatisfiable. We give the full algorithm and a proof of correctness in [**?**].

This algorithm has certain advantages over other algorithms proposed for CLP($\Re$). First, the operations of Gaussian elimination and the simplex method are cleanly separated, making the algorithm simple to describe, implement, and prove correct. The clean dichotomy also increases the algorithm's efficiency in that only the all-slack rows are involved in the simplex procedure. Furthermore, only the forward elimination part of Gaussian elimination is used to uncover linear dependencies in the tableau, back substitution being delayed until the end of a successful solution path. Forward elimination is often sufficient for the satisfiability check, and we are saved the expense of back substitution (which serves only to make the solution more explicit) on failure paths. Most importantly for our purposes here, we can show that this algorithm naturally generates minimal conflict sets.

[3] For efficiency reasons, variables are ordered according to their time of creation, newer variables placed before older ones, and program variables before slack variables. To avoid confusion, we do not show the newer-to-older ordering in the examples below.

4 Collecting Conflict Sets

To identify a conflict set when *checksat* uncovers a conflict, we keep a record of row operations. This process can easily be described with a matrix representation. Without loss of generality, we can assume that the solver is given m constraints to process, and it processes them one at a time, finding a conflict when it gets to the m^{th} one. R is an $m \times n+1$ matrix representing the original forms of the rows in the tableau. M is a dynamic $m \times n+1$ matrix representing the rows as they change state during *checksat*. We will use an $m \times m$ matrix B to record the row operations which transform R to M. Initially, B is the identity matrix. Each time we add a multiple of one row to another during Gaussian elimination or simplex, as in $M_j \leftarrow c * M_i + M_j$ we also perform $B_j \leftarrow c * B_i + B_j$. By this means, we maintain the relation $M = B * R$. In particular, say that the last row of B is $[B_{m,1}, ..., B_{m,m}]$. Then we have the relation

$$M_m = B_{m,1} * R_1 + ... + B_{m,m} * R_m \tag{1}$$

(where $B_{m,m}$ is either 1 or -1). When a conflict is uncovered in M_m, we claim the set of rows $C = \{R_i \mid B_{m,i} \neq 0\}$ constitutes a conflict set.[4]

Figure 3 illustrates how the source of conflict can be identified by our method. Note that the i^{th} program variable to be encountered during program execution (not including variables eliminated when redundancies or conflicts are uncovered) is renamed X_i at the time of creation. Note also

[4]Implicitly, the conflict set C also contains the inequality $s_i \geq 0$ for every slack variable s_i in the rows in C.

that unifications are entered as equations into the tableau. When the conflict is uncovered here, B_9 identifies M_9 as a linear combination of rows R_9, R_8, R_7, R_6, R_5, R_4, R_2, and R_1. The sources of these rows are nodes 4, 2, and 1, indicating that node 3 had nothing to do with the conflict. Thus, we can backtrack directly from node 4 to node 2.

5 Minimal Conflict Sets

We now show that the conflict set C identifiable through B is minimal; that is, no proper subset of C is also a conflict set. In the following, we will treat an equation, say $q_1x_1 + \ldots + q_nx_n = b$, as a row vector $[q_1, \ldots, q_n, b]$, and also as an algebraic expression $b - (q_1x_1 + \ldots + q_nx_n)$. The following lemma summarizes the end result in the tableau when a conflict is uncovered.

Lemma 5.1 *Suppose that during* checksat*, the current tableau* $Q = \{M_1, \ldots, M_{m-1}\}$ *is in solved form and is consistent, but the new row in its original form, denoted* R_m*, is found to be inconsistent with* Q *when* R_m *is transformed to its current form* M_m*. Then, writing* M_m *as an expression,* $M_m = M_{m,1}x_1 + \ldots + M_{m,n}x_n + b$*, the following holds: If the inconsistency is found during the forward elimination step, then all* $M_{m,i} = 0$*,* $1 \leq i \leq n$*, and* $b > 0$*; otherwise, if the inconsistency is found during simplex, then* $b > 0$*,* $M_{m,i} \geq 0$ *for all* $1 \leq i \leq n$*, and for all* $M_{m,i} > 0$*, the corresponding variable* x_i *must be a slack variable.*

As a result of Lemma 1, when the algorithm *checksat* uncovers a conflict in row M_m in either case, we have

$$M_m = \sum_{s_i \in D} c_i s_i + b \tag{2}$$

where $c_i > 0$, D is either empty or contains slack variables only, and $b > 0$.

We illustrate equations (1) and (2) using an example in which a conflict is found during simplex. Consider the following tableau.

$$\begin{array}{ll} R_1 : X_1 \geq 1 & R_5 : X_5 \leq 1 \\ R_2 : X_2 \geq 1 & R_6 : X_1 + X_2 + X_3 = 5 \\ R_3 : X_3 \leq 1 & R_7 : X_3 + X_4 = 1 \\ R_4 : X_4 \leq 1 & R_8 : X_1 + X_5 = 7 \end{array}$$

The five inequalities are converted to equations with slack variables. This tableau shows the resulting equations, with basic variables identified by $*$.

	X_1	X_2	X_3	X_4	X_5	S_1	S_2	S_3	S_4	S_5	b
R_1	1^*	0	0	0	0	-1	0	0	0	0	1
R_2	0	1^*	0	0	0	0	-1	0	0	0	1
R_3	0	0	1^*	0	0	0	0	1	0	0	1
R_4	0	0	0	1^*	0	0	0	0	1	0	1
R_5	0	0	0	0	1^*	0	0	0	0	1	1

To process R_6, we substitute out basic variables X_1, X_2, and X_3, The resulting all-slack row is kept in a separate tableau. Simplex chooses S_2 as a basic variable in it. Similarly, when R_7 is processed, the basic variables X_3 and X_4 are substituted out, resulting in an all-slack row $-S_3 - S_4 = -1$. Then, negating both sides yields the row $S_3 + S_4 = 1$. The following shows the tableau of all-slack rows with the basic variable identified.

	S_1	S_2	S_3	S_4	S_5	b
M'_6	1	1^*	-1	0	0	2
M'_7	0	0	1	1^*	0	1

At this point, the equations R_1 through R_7 are consistent. To process R_8, we first substitute out the basic variables X_1 and X_5, resulting in the all-slack row M'_8 added to the tableau.

	S_1	S_2	S_3	S_4	S_5	b
M'_6	1	1^*	-1	0	0	2
M'_7	0	0	1	1^*	0	1
M'_8	1	0	0	0	-1	5

To determine the consistency of the system of R_1 through R_8, we need to apply simplex to the all-slack rows. The essence of the simplex procedure is to treat M'_8 as an objective function to be minimized, given the solution space determined by M'_6 and M'_7. When the standard pivoting procedure is applied twice, the tableau yields a minimized objective function $M_8 = 2+S_2+S_4+S_5$.

	S_1	S_2	S_3	S_4	S_5	b
M_6	1^*	1	0	1	0	3
M_7	0	0	1^*	1	0	1
M_8	0	-1	0	-1	-1	2

Keeping track of the row operations, we obtain the following algebraic identity, treating each row ($LHS = RHS$) as an expression, $RHS - LHS$.

$$\begin{aligned} M_8 &= -M'_6 - M'_7 + M'_8 \\ &= -(-R_1 - R_2 - R_3 + R_6) - (R_3 + R_4 - R_7) + (-R_1 - R_5 + R_8) \\ &= R_2 - R_4 - R_5 - R_6 + R_7 + R_8 \\ &= 2 + S_2 + S_4 + S_5 \end{aligned}$$

As noted in (1), M_m is a linear combination of rows in their original form. Combining (1) and (2) yields

$$M_m = \sum_{R_i \in C} B_{m,i} R_i = \sum_{s_i \in D} c_i s_i + b \tag{3}$$

Since any solution satisfying the equations $R_i \in C$, when substituted into (3), would yield

$$0 = \sum_{s_i \in D} c_i s_i + b > 0,$$

clearly the set $C = \{R_i | B_{m,i} \neq 0\}$ forms a set of conflicting equations. It is obvious that C contains equation R_m, because $\{R_1, \ldots, R_{m-1}\}$ is known to be satisfiable. To prove that the set is a minimal conflict set, we first note that since (3) is an algebraic identity, those slack variables s_i appearing on the righthand side must also appear on the lefthand side. Since a unique slack variable is introduced from its "source" inequality, the slack variables on the righthand side of (3) identify exactly those inequalities whose corresponding equations appear on the lefthand side of (3). We now prove that C forms a minimal conflict set.

Theorem 5.2 *Given a satisfiable tableau of $m-1$ rows, if algorithm* check-sat *uncovers a conflict when it processes a new row R_m, then*
$C = \{R_i | B_{m,i} \neq 0\}$ constitutes a minimal conflict set.

Proof: Suppose that C is not minimal. That is, there exists a set C' which is a proper subset of C, where C' is also a conflict set. The set C' must contain R_m because the previous $m-1$ rows $R_1, \ldots, R_{m-1}$ are satisfiable. Applying the algorithm *checksat* to C' will lead to a relation similar to (3), according to Lemma 1:

$$\sum_{R_i \in C'} B'_{m,i} R_i = \sum_{s_i \in D'} c'_i s_i + b' \qquad (4)$$

Without loss of generality, we assume that in both (3) and (4), the coefficient for R_m on the lefthand side is 1. Combining (3) and (4) yields

$$\begin{aligned} R_m &= -\sum_{R_i \in C - \{R_m\}} B_{m,i} R_i + \sum_{s_i \in D} c_i s_i + b \\ &= -\sum_{R_i \in C' - \{R_m\}} B'_{m,i} R_i + \sum_{s_i \in D'} c'_i s_i + b' \qquad (5) \end{aligned}$$

The tableau $\{M_1, \ldots, M_{m-1}\}$ is in solved form when a conflict with M_m is uncovered; thus, the basic variables in $\{M_1, \ldots, M_{m-1}\}$ can be solved in terms of the non-basic variables. Substituting these solutions for the basic variables into (3), using the fact that $R = B^{-1}M$, all the expressions R_i, $1 \leq i \leq m-1$, would vanish. Thus, after substitution, equation (3) becomes

$$R_m{}^* = \sum_{s_i \in D} c_i s_i + b$$

where $R_m{}^* = (R_m$ after substitutions).

Note that the expression

$$\sum_{s_i \in D} c_i s_i + b$$

in (5) is not affected by the substitutions because the variables in D are non-basic. Similarly, the same set of substitutions into (4) yields

$$R_m{}^* = \sum_{s_i \in D'} c'_i s_i + b'$$

because these same substitutions erase all R_i's, $1 \leq i \leq m-1$. Therefore, we have

$$\sum_{s_i \in D} c_i s_i + b = \sum_{s_i \in D'} c'_i s_i + b' \tag{6}$$

Since (6) is an algebraic identity, the two sides must be identical expressions. Thus $D = D'$, $c_i = c'_i$, for each $s_i \in D$, and $b = b'$.

Substituting (6) into (5) yields

$$\sum_{R_i \in C - \{R_m\}} B_{m,i} R_i = \sum_{R_i \in C' - \{R_m\}} B'_{m,i} R_i$$

which results in a non-trivial dependency among the rows $R_i \in C$, because we assumed C' is a proper subset of C. This dependency relation contradicts the fact that there are no redundant rows in C, because there are no redundant rows in the tableau $\{M_1, \ldots, M_m\}$. This contradiction proves the theorem.

6 A Two-Sorted Domain

Some implicit assumptions were made in Figure 3 in order to simplify the initial discussion of our backtracking method. First, we assume that variable bindings are entered as equations in the tableau. If binding information is not integrated into the collection of conflict sets, solutions may be missed. In our example, $U + V = 5$ from clause p is in conflict with $U + 2V = 8$ from clause r, but only in the context of the bindings equating the original U and V, which occurred when goal q was executed. The presence of equations R_5 and R_6 in the conflict set ensures that node 2 is in the backtrack set, as it should be. A second simplifying assumption is that all variables are type *real*, so there is no need to check for type clashes, nor to account for the recursive unification of arguments to structured terms. In this section, we abandon this second assumption and deal with the two-sorted domain, temporarily retaining the method of recording variable bindings as equations.

To handle variables of two types in CLP($\Re$), we define an abstract supertype *term*, of which types *structured term* and *real number* are subtypes. Equality between terms is defined as follows:

If t is a structured term and c is a real number, then $t \neq c$. (An attempt to unify terms of different types is called a *type clash.*)

If t and u are structured terms and $t = u$, then their function symbols are identical, t and u both have n arguments ($n \geq 0$) and for $1 \leq i \leq n$, $v_i = w_i$, where v_i is the i^{th} argument of t, and w_i is the i^{th} argument of u.

Equality over real numbers is defined in the usual manner.

Since bindings are handed to the solver as equations, we now have equations over terms in the tableau. Each unification equation $v_i = w_i$ created between the i^{th} pair of arguments to two unified structured terms is called a *recursive unification equation*. Equations and inequalities found in the body of a clause are referred to as *clause equations*.

We propose algorithm *enterclause* to generalize the work of the solver to a two-sorted domain. This algorithm initially processes all unifications and clause equations as equations over terms. *enterclause* uses a generalized process of forward elimination, making our method for collecting conflict sets directly applicable. Forward elimination is now divided into three stages. The first stage, applicable to unification equations, dereferences variables by following binding chains. The second and third stages are applied to equations with real number expressions on both sides after dereferencing, or clause equations which are input in this form. In the second stage, all variables are identified as type *real*. The third stage continues forward elimination in the domain of reals using *checksat*, applying simplex if necessary.

In all three stages, we maintain R, M, and B as before. It should be clear that in the simple case, where no recursive unification equations are involved, our method of collecting backtrack sets generalizes directly to the two-sorted domain. Consider the case where a type clash is uncovered during dereferencing. As long as we are processing equations over terms, we can view each structured term or real number expression simply as a constant. Say that X_1 is bound to 3, X_2 is bound to $g(X_3)$ and then we attempt to bind X_2 to X_1. This situation is represented by the following tableau:

R	M	B
$X_1 = 3$	$X_1^* = 3$	R_1
$X_2 = g(X_3)$	$X_2^* = g(X_3)$	R_2
$X_2 = X_1$	$g(X_3) = 3$	$R_3 - R_2 + R_1$

By our usual method of recording row operations, the conflict set is identified as $\{R_1, R_2, R_3\}$. In the case where an equation consists entirely of real number expressions, extending our method for recording row operations from the dereferencing step to the *checksat* algorithm is also straightforward.

We now must account for the presence of recursive unification equations in the tableau. Since a recursive unification equation can be introduced as a result of binding chains, we are interested in the *history* of such an equation, identifying nodes where bindings which eventually entailed the equation can be undone. Consider the example in Figure 4, where $f(X_2) = f(X_1)$ is the result of forward elimination on R_{11}. Since $f(X_2) = f(X_1)$ results from the linear combination of R_{11}, R_{10}, R_8, R_5, and R_4, these rows make up the history of the recursive unification equation $X_2 = X_1$.

More generally, the history of the m^{th} row, $HIST(R_m)$, tells us the constraints which brought R_m into existence. Say that R_m is a recursive unification equation which arose when two structured terms were unified in M_i. Let $BROWS(i)$ denote $\{R_k | B_{i,k} \neq 0\}$. Then $HIST(R_m)$ is defined as

$$HIST(R_m) = \bigcup_{R_j \in BROWS(i)} HIST(R_j).$$

If R_m is a non-recursive unification equation or a clause equation, then $HIST(R_m) = \{R_m\}$. If a new row R_m is found to be inconsistent with the previous $m-1$ rows, then we identify a conflict set containing R_m as

$$CONFL(R_m) = \bigcup_{R_j \in BROWS(m)} HIST(R_j).$$

If R_m is not in conflict with the rest of the tableau, $CONFL(R_m) = \{\}$.

It should be noted that $CONFL(R_m)$ is not necessarily minimal. (See [?] for an example.) However, it suffices to identify *any* conflict set to apply the DIB algorithm without missing a solution, and a significant degree of intelligent backtracking can be achieved without minimality.

7 A Generator-Consumer Approach

We have shown how our method for collecting conflict sets can be extended to account for structured terms. This method is based upon the recording of unifications as equations in the tableau, an admittedly inefficient strategy. We now offer a more efficient approach arising from the following observation.

Let the *ancestor path* of a goal literal p be defined as an ordered set of node numbers $\{n_0, n_1, \ldots, n_m\}$ where n_0 is the number of the node where p is the leftmost goal, and for $1 \leq i \leq m$, n_i is the node number of the source of the leftmost literal at node n_{i-1}. If the source of the leftmost literal at node n_i is 0, then $m = i+1$. The *source of a variable* which is created when goal p is executed is the node where goal p is leftmost. The *ancestor path of a variable* is the ancestor path of the goal literal being executed when the variable is created. Consider the example in Figure 5. Here, the source of X_5 is node 7 and its ancestor path is $\{7,3,1\}$. Equations $X_1 + X_2 = 5$ and $X_5 + X_6 = 6$ are in conflict, but only in the context of the bindings which occurred at nodes 7 and 3. By our previous method, the equations representing the bindings would be part of the conflict set. However, putting the source of these equations in the backtrack set introduces redundant information. If we fail on all paths from node 7, the DIB algorithm requires that we place the source of goal p, node 3, in the backtrack set. If we fail at node 3, the source of the goal q, node 1, is placed in the backtrack set. As failure continues, we will always backtrack through the ancestor paths of the failing goals, precisely the nodes associated with the input bindings. Thus, we need only record extra "stopping places" along the way so as not to miss a solution. That is, we need to mark a variable binding only if that binding occurs at some node after the node where the variable was created. Such a situation is illustrated by Figure 3, where we need to mark the binding of X_2 to X_1 as having occurred at node 2.

By this reasoning, we can simplify our method for tracing bindings. Let us represent variable bindings by pointers through the variable space rather than equations. Using a dereferencing procedure analogous to our forward elimination procedure in *checksat*, we follow the binding chains of the variables being unified, and if there is at least one unbound variable at the ends of these chains, we bind the newer variable to the other term.

Analogous to our method of recording row operations during Gaussian elimination, we trace the reasons for each variable X_i's binding in a generator set (denoted $GEN(X_i)$), that is, a set of nodes where alternative paths can be tried in order to undo the binding. For any variable X_i, if X_i receives its binding at the node where it is first created, then $GEN(X_i)$ is the empty set. Otherwise, let p be the goal being executed when X_i receives its binding, let node n be the node where p is leftmost, and let V be the set of variables whose dereferencing led to the binding of X_i. In the case where X_i's binding is being changed during execution of some node after the node where X_i was created,

$$GEN(X_i) = (\bigcup_{X_k \in V} GEN(X_k)) \cup \{n\}$$

Thus, we collect generator sets during dereferencing and store the collected set with the variable which is eventually bound, if a binding is made. If dereferencing of a unification leads us to two structured terms requiring recursive unifications, we carry the generator set along with each recursive unification and continue the dereferencing in the same manner. If dereferencing eventually uncovers a conflict, the generator set thus collected tells us nodes not on the ancestor path of the literal being executed, but where we can try another path of execution with the possibility of finding a solution.

In Figure 6, X_2 is bound to X_1 when r is executed . Since X_2's binding is being changed, it is marked with the generator set $\{4\}$, containing only the source of r. When X_{10} is unified with 2 at node 6, we use the binding of X_2 to X_1 in the dereferencing, and the generator set is $\{4\} \cup \{6\} = \{4, 6\}$. We find a conflict when we later attempt to unify X_1 with 3, and the generator set becomes $\{4, 6, 8\}$, including the source of the goal just executed.

This method takes a generator-consumer approach to intelligent backtracking, similar to [**?**], by identifying nodes which generate bindings for earlier nodes. Our method optimizes the generator-consumer approach by recognizing that bindings generated for a node by its "ancestor" nodes need not be recorded, since the ancestor nodes will automatically be included in the backtrack set by the DIB algorithm. We can integrate this approach into CLP($\Re$)'s constraint satisfaction algorithm by modifying our definition of the history of a clause equation so that it consists of node numbers rather than rows. When an equation is being prepared for the solver, the variables therein are dereferenced in the manner described above. Let V be the set of all the variables dereferenced when the equation is prepared for the solver. Then the history for the equation when it enters the tableau consists of

$$\bigcup_{X_k \in V} GEN(X_k) \cup \{n\}$$

where n is the node number of the source of the equation. With this definition for the history of an equation, we can again use B to identify a backtrack set when a conflict appears. If row R_m is found to be inconsistent with a satisfiable tableau of $m-1$ rows, then we identify the backtrack set associated with the conflict as

$$\bigcup_{R_j \in BROWS(m)} HIST(R_j).$$

8 Implementation

We have implemented a CLP($\Re$) interpreter in C using *checksat*. The tableau is stored in R and a working copy of the solved form is maintained in M, with row operations recorded in B and conflict sets thereby generated. In a run of the Figure 1 program, equations 19, 14, 12, 11, and 10 are correctly identified as constituting a conflict set at node 15, while equations 20, 14, 13, 12, 11, 10, and 9 constitute the conflict set at node 17. We have also installed the DIB algorithm into the inference engine's backtracking mechanism and integrated the identification of conflict sets between the solver and the inference engine. Tests runs on the canonical cryptarithmetic problem show a 14% speedup despite the overhead. We note that the cost of our intelligent backtracking mechanism is reasonable. Only matrix B is required for storing row operations. If we further optimize the algorithm by employing the revised simplex method, we get minimal conflict sets at no added expense, since in the revised simplex method the tableau is represented by the operations performed on it rather than by its current state [**?**]. The tracing of variable bindings requires only (1) a comparison of the node number of the variable being bound to the current node number, (2) the storing of a generator set for each variable, and (3) the unioning of generator sets during dereferencing. In many cases, the generator sets will be empty. In the Figure 1 example, a small amount of extra bookkeeping results in a significant benefit in intelligent backtracking, since each of nodes 14, 12, 11, 10, $9''$, $8'$, 6 and $5'$ is a choice point. We conclude that our constraint satisfaction algorithm and technique for tracing variable bindings combine to form a feasible intelligent backtracking algorithm for the two-sorted domain of CLP($\Re$).

References

[1] Bruynooghe, M., and L. Pereira. Deduction revision by intelligent backtracking. In J. Campbell, ed. *Implementations of Prolog*, Ellis Horwood, 1984.

[2] Burg, J. Parallel execution models and algorithms for constraint logic programming over a real-number domain. PhD Dissertation, University of Central Florida, December 1992.

[3] Burg, J., S.-D. Lang, and C. Hughes. Finding conflict sets and backtrack points in CLP($\Re$). Technical Report TR-CS-93-01, Wake Forest University, Winston-Salem, NC. Nov. 1993.

[4] Codognet, C., P. Codognet, and G. Filé. Yet another intelligent backtracking method. *Proc. of the Fifth Int. Conf. and Symp. on Logic Programming*, (Seattle, 1988): 447-465.

[5] Codognet, P. and T. Sola. Extending the WAM for intelligent backtracking. *Proc of the Eighth Int. Conf. on Logic Programming*, (Paris, France, 1991): 127-141.

[6] Cox, P. Finding backtrack points for intelligent backtracking. In J. Campbell, ed. *Implementations of Prolog*, Ellis Horwood, 1984.

[7] DeBacker, B. and H. Beringer. Intelligent backtracking for CLP languages: An application to CLP($\Re$). *Proc. of the international symposium on logic programming.* (San Diego, Oct. 1991), 405-419.

[8] Heintze, et al. The CLP($\Re$) programmer's manual, Version 1.2. IBM Thomas J. Watson Research Center, Sept. 1992.

[9] Hogger, C., and A. Kotzamanidis. Aspects of failure analysis in a CLP($\Re$) system. Internal Report, Imperial College, London, June 1993.

[10] Jaffar, et al. An abstract machine for CLP($\Re$). *Proc. of SIGPLAN 92 conference on programming language description and implementation.* (San Francisco, May 1992), 128-139.

[11] Jaffar, et al. The CLP($\Re$) language and system. *ACM transactions on programming languages and systems* 14, 3 (July 1992), 339-395.

[12] Kumar, V., and Y.-J. Lin. An intelligent backtracking scheme for Prolog. *Proc. of the international symposium on logic programming.* (Sept. 1987), 406-414.

[13] Luenberger, D. *Linear and Nonlinear Programming.* 2nd ed. Reading, Mass.: Addison-Wesley.