Constraint Management in the ATTEST System

Laura K. Dillon

COINS Technical Report 81-9
May 1981

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Air Force Office of Scientific Research under grant AFOSR 77-3287.

Table of Contents

1. Overview of CONMAN 1

2. Representation of the Partial Path Condition 6
   2.1 Path Constraints 6
   2.2 Conceptual Representation 14
   2.3 Constraint Transformation 19
   2.4 Sparse Array Representation 24

3. Maintaining the Partial Path Condition 30

4. Analysis of the Partial Path Condition 38
   4.1 Overview of TRYPTH 38
   4.2 Description of TRYPTH 45

5. Redundancy checking 60

6. Reconfiguration 64
   6.1 Equivalence Classes 66
   6.2 The Reconfiguation Algorithm 70

7. Reduction of the Constraint Structure 77

8. Detecting Possible Program Errors 83

Appendix A: The BNB structure 86

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER University of Massachusetts COINS Technical Report TR81-9	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Constraint Management in the ATTEST System		5. TYPE OF REPORT & PERIOD COVERED Final
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Laurie K. Dillon		8. CONTRACT OR GRANT NUMBER*(s)* AFOSR 77-3287
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer and Information Science Department University of Massachusetts Amherst, Massachusetts 01003		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research Washington, D.C.		12. REPORT DATE October 1981
		13. NUMBER OF PAGES 91
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Symbolic Evaluation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

ATTEST is a symbolic execution system designed to be used as a tool for validating programs. This report describes the constraint management component (CONMAN) of the ATTEST system. This component maintains a consistent representation of the constraints created during symbolic execution.

DD FORM 1 JAN 73 1473

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

Table of Figures

Figure 1 -
Sample FORTRAN Program 8

Figure 2 -
Graph for the Sample Program of Figure 1 9

Figure 3 -
The Successive Partial Path Conditions That Are Generated During Symbolic Execution of P2 12

Figure 4 -
Matrix Conceptualization of the AND-structure After Symbolic Execution of P2 16

Figure 5 -
Matrix Conceptualization of the OR-structure After Symbolic Execution of P2 18

Figure 6 -
The AND-constraint Generated by P2 Shown in Standard Form and After Being Transformed by CONMAN 22

Figure 7 -
Conceptualization of the Sparse Array Representation Used in the AND-structure 25

Figure 8 -
Sparse Array Representation of the Matrix of Coefficients of the Transformed AND-constraints of Figure 6 26

Figure 9 -
Sparse Array Representation of the OR-matrix of Figure 5 28

Figure 10 -
Partitioning of the Constraint Structure into New and Old Sections 33

Figure 11 -
Evolution of the Constraint Structure as the First Four Branch Predicates are Encountered During Symbolic Execution of P2 35

Figure 12 -
Constraint Structure Immediately Before (top) and After (bottom) Detection of an Infeasible Path 37

Figure 13 -
Summary of PROANS' Algorithm 47

Table of Figures (continued)

Figure 14 -
Summary of PROGP's Algorithm 52

Figure 15 -
Analysis of the Path P2 56

Figure 16 -
Redundancy Table 62

Figure 17 -
Partition of Constraint Structure into two Equivalence Classes 68

Figure 18 -
The Reconfiguration Algorithm 72

Figure 19 -
A Sample Reconfiguration 75

Figure 20 -
Division of the BNB Structure into its two Components: AND-structure and Effective OR-clause Structure 87

1. Overview of CONMAN

ATTEST is a symbolic execution system designed to be used as a tool for validating programs. This report describes one component of the ATTEST system, CONMAN, which maintains a consistent representation of the constraints that are created during symbolic execution. Before an overview of CONMAN is given, an overview of ATTEST and some terminology are presented.

Symbolic execution is a program validation method that produces symbolic representations for the computations and domain of a program path. To symbolically execute a program path, symbolic names are substituted for the input values of the path and each statement on the path is interpreted using these symbolic names. The result of symbolically executing a statement along a path depends on the nature of the particular statement. For example, symbolically executing a read statement results in the input variables being assigned unique symbolic names to represent the input values. Symbolically executing an assignment statement that assigns the value of an arithmetic computation to a variable, results in the variable being assigned an algebraic expression that represents the computation. The algebraic expression generated by an assignment statement is in terms of the symbolic names that have been assigned to represent the input values. The symbolic representations of a path's computations are provided by the algebraic expressions for the path's output values.

The domain of a program path is determined by the conditional statements that occur on the path. A symbolic representation for this domain can be obtained by symbolically executing these conditional statements as they occur on the path. Symbolically executing a conditional statement results in a symbolic representation of the condition that must be satisfied if execution is to proceed along the branch specified by the path. This symbolic representation is a logical expression of equalities and inequalities in the symbolic names of the input values and is called the _branch condition_. After symbolic execution of a path, the branch conditions of the executed conditional statements can be conjoined into one logical expression, called the _path condition_. The path condition describes, in terms of the symbolic names of the input values, precisely those input values that would cause execution of the selected path, thereby providing a symbolic representation for the domain of the path. If a solution exists to the path condition, the path condition is said to be _consistent_ and the path is _executable_ or _feasible_. Conversely, the path is _nonexecutable_ or _infeasible_ and the path condition is _inconsistent_ if there is no assignment of values to symbolic names that would satisfy the path condition.

The ATTEST system symbolically executes a program path and then uses the path condition to determine path feasibility, to create test data for feasible paths and to detect certain program errors. ATTEST uses a linear inequality solver to determine path feasibility and create test data. ATTEST is therefore limited in these determinations to paths whose path conditions can be represented by linear equalities and inequalities.

ATTEST constructs the path condition incrementally. Thus, as it symbolically executes a path, it maintains the partial path condition for that portion of the path that has been executed. A partial path condition is defined by the conjunction of the branch conditions that have been generated by symbolic execution of an initial segment of a program path, or, if no branch conditions have been generated, by the constant logical expression TRUE.

When a branch condition is created, it is checked for consistency with the partial path condition that already exists. If consistent, it is conjoined to the existing partial path condition and symbolic execution of the path continues. Otherwise, the path is infeasible. In this case, either symbolic execution is terminated altogether or the path to be executed is redefined so that symbolic execution continues down an alternate branch associated with the conditional statement.

In addition to determining the feasibility of a path and to generating test data, the ATTEST system determines if certain program errors can occur along the path. An error condition, such as array index out of bounds or division by zero, can be represented symbolically by a logical expression of equalities or inequalities in the symbolic names of the input values of the path. When the possibility of one of these program errors is detected during symbolic execution of the path, a symbolic representation of the error condition, called a temporary condition, is generated. The temporary condition is checked for consistency with the existing partial path condition. If consistent, a warning message is returned to the user and a data set is generated that would cause the error. Otherwise, the error cannot occur. In either case, the temporary condition is not conjoined to the partial path condition and path execution is continued.

This report describes the constraint management component (CONMAN) of the ATTEST system. CONMAN builds and analyzes the path condition of a selected path to determine if the path is executable. When the path is executable CONMAN also generates test data that would drive execution down the path. The partial path condition is maintained in a data structure, called the constraint structure. CONMAN checks each new branch condition for redundancies with the existing partial path condition when it adds a new branch condition to the constraint structure. It then determines if the new branch condition is consistent with the existing partial path condition. In some cases, consistency can be determined from the results of the redundancy check. In other cases, the inequality solver is used in an attempt to generate a solution that satisfies the partial path condition recorded in the constraint structure. Based on these determinations, CONMAN makes appropriate reductions to the constraint structure.

Section 2 of this report describes how the constraint structure is organized. Section 3 outlines the manner in which CONMAN maintains the constraint structure to represent the partial path condition. In section 4 the procedure by which the partial path condition is tested for consistency is described. The process that detects redundancies and inconsistencies between the equalities and inequalities in the path condition is described in Section 5. The reconfiguration process, which selects alternative clauses from disjunctive branch conditions, is described in Section 6. The reductions that CONMAN makes to the constraint structure are outlined in section 7. Finally, section 8 describes how temporary conditions are used to detect possible program errors. The design for CONMAN appears in COINS Technical Note TN/CS/00047.

2. Representation of the Partial Path Condition

CONMAN maintains the constraint structure to represent the partial path condition for that portion of the path that has been symbolically executed. This section describes this representation. The first subsection defines the fundamental units, called path constraints, that are used in this representation. An intermediate conceptualization of the constraint structure is described in the second subsection. The transformations that the path constraints undergo before being represented in the constraint structure are discussed in the third subsection, and finally, the actual representation of the path constraints is presented in the last subsection of this section.

2.1. Path Constraints

To facilitate the analysis, the partial path condition is maintained in conjunctive normal form. Thus, the partial path condition is composed of the conjunction of a finite number of logical subexpressions, each of which is either a single equality or inequality or the disjunction of equalities and inequalities. Each subexpression is called a _path constraint_. Each path constraint is classified according to whether it is a disjunction of equalities and inequalities, called an _OR-group_, or whether it is a single equality or inequality, called an _AND-constraint_. The individual equalities and inequalities of an OR-group are called _OR-clauses_. The conjunction of the path constraints that are generated by symbolic execution of an initial segment of a program path defines the domain of the partial path. Thus, all of the AND-constraints and at least one OR-clause from each OR-group must be satisfied if the partial path is to be executed.

Representations for the path constraints are recorded in the constraint structure as the path constraints are generated from symbolically executing conditional statements. A non-empty constraint structure represents the logical condition obtained by conjoining the path constraints that are recorded in it, while an empty constraint structure represents the constant logical condition TRUE.

Paths from the sample program of Figure 1 are used to illustrate the representations of path constraints. A graph for this program is given in Figure 2. Conditional statements have been represented in the program graph by expressions, called _branch predicates_, that annotate the associated edges in the graph that correspond to branches in the program. Each branch predicate represents the condition that must be satisfied by the values of the program variables if the branch associated with the edge is to be executed.

Nodes in the graph and corresponding statements of the program have been numbered for easy reference. A program path can be specified by the sequence of node numbers that are encountered along the path. Symbolic execution of this path can then be viewed as the process of symbolically

```
                         PROGRAM MAXMAG

 C   GIVEN A POSITIVE INTEGER N AND A SEQUENCE OF N
 C   REAL NUMBERS, MAXMAG RETURNS THE MAXIMUM
 C   MAGNITUDE OF THE NUMBERS IN THE SEQUENCE

          INTEGER N, I
          REAL    A, M

  1       READ N
  2       IF    (N .LE.  0)
         +        GOTO 100
  3       M = 0.0
  4       I = 1
  5  200  READ A
  6       IF   (A .LE.  M .AND.  -A .LE.  M)
         +        GOTO 300
  7       IF    (A .GT.  M)
  8      +        M = A
  9       IF    (-A .GT.  M)
 10      +        M = -A
 11  300  I = I + 1
 12       IF    (I .LE.  N)
         +        GOTO 200
 13       WRITE M
 14       STOP
 15  100  WRITE "ERROR"
 16       STOP
          END
```

Figure 1

Sample FORTRAN Program

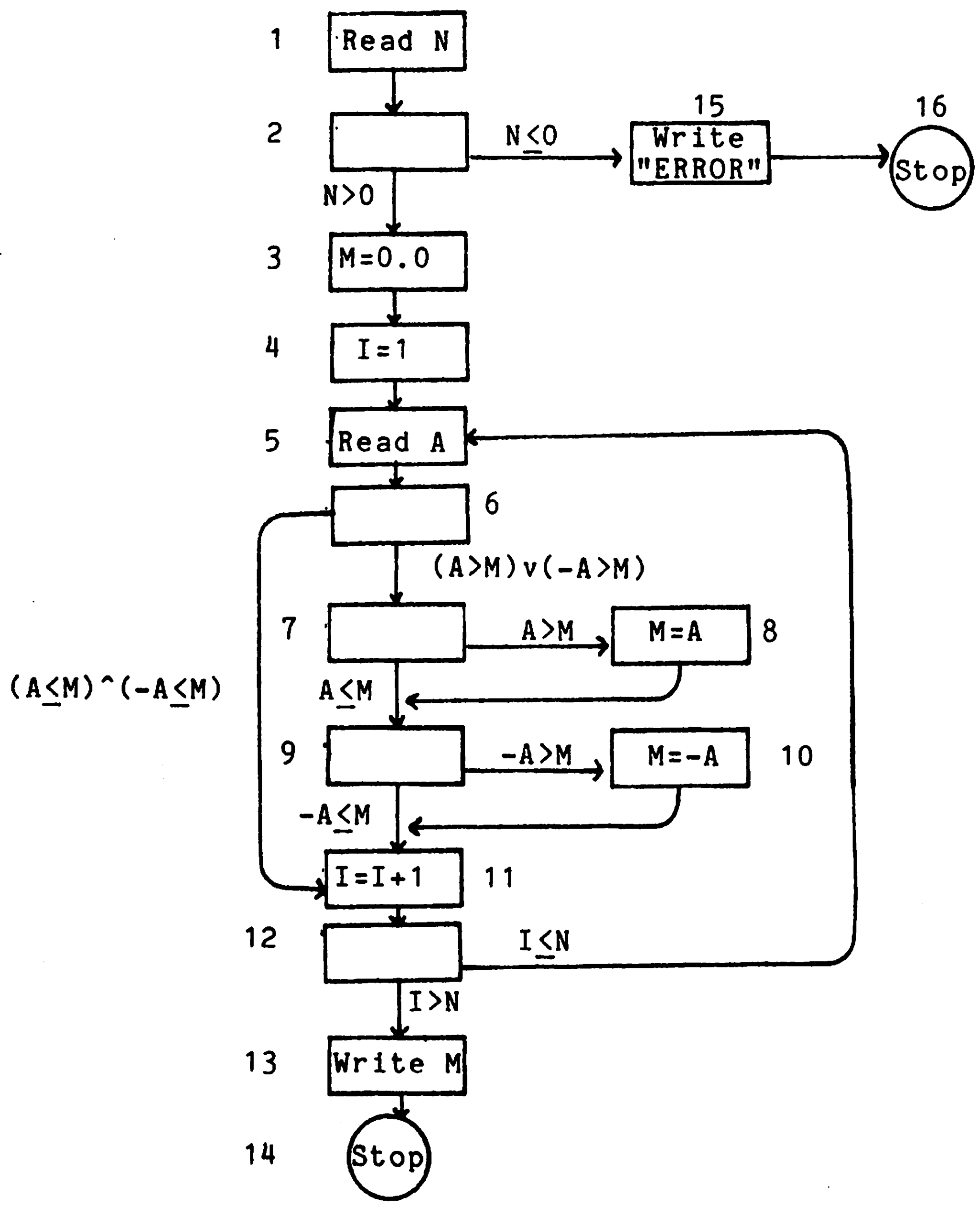


Figure 2

Graph for the Sample Program of Figure 1 executing the nodes and the edges that join the nodes in this sequence. Symbolically executing the nodes generates the symbolic representations for the program variables. Symbolically executing the edges generates the logical expressions that are conjoined into the path condition. A branch condition is generated by symbolic execution of an annotated edge, while the constant logical expression TRUE is generated by symbolic execution of an edge that is not annoted with a branch predicate.

Given a program path, Pk, Pk\n denotes the partial path determined by the first n nodes in the sequence that specifies Pk, and Pk\n+ denotes the extension of the partial path Pk\n to include the edge joining node n and node (n+1) of the sequence. For example, let P1 specify the program path (1 2 15 16). Then P1\2 denotes the partial path (1 2), and P1\2+ denotes the extension of the path (1 2) to include the edge (2,15). Note that node 1, edge (1,2) and node 2 are executed during symbolic execution of P1\2, whereas the path P1\2 and then the edge (2,15) are executed during symbolic execution of P1\2+. The path condition for P1 is obtained by conjoining the logical expressions generated by symbolically executing the edges (1,2), (2,15) and (15,16). The edges (1,2) and (15,16) both generate the logical expression TRUE. The edge (2,15) generates a branch condition that can be represented by the inequality

$$(S(N) \leq 0),$$

where S(N) denotes the symbolic representation of N after execution of P1\2. Using the notation "variable-name$k" to denote the kth symbolic name assigned to the variable "variable-name", the above inequality results in the following representation for the branch condition:

$$(N\$1 \leq 0).$$

Thus, the path condition obtained by symbolically executing P1 is represented by the logical expression

$$((TRUE) \text{ and } (N\$1 \leq 0) \text{ and } (TRUE)),$$

which simplifies to $(N\$1 \leq 0)$. Since a path condition trivially simplifies to the conjunction of the branch conditions obtained by symbolically executing the annotated edges in the path, the unannotated edges are not discussed further.

For a less trivial example consider the path P2 = (1 2 3 4 5 6 7 8 9 11 12 5 6 11 12 5 6 7 9 10 11 12 13 14). Symbolic execution of the partial path P2\2+ results in the partial path condition $(N\$1 > 0)$. Since edge (6,7) represents the next conditional statement in P2, the partial path condition remains constant until this edge is encountered. The branch condition generated by the edge (6,7) is $((A\$1 > 0.0) \text{ or } (-A\$1 > 0.0))$. Thus, the partial path condition for the partial path P2\6+ is:

$$((N\$1 > 0) \text{ and } ((A\$1 > 0.0) \text{ or } (-A\$1 > 0.0))).$$

This partial path condition consists of two constraints: one AND-constraint, $(N\$1 > 0)$; and one OR-group, $((A\$1 > 0.0) \text{ or } (-A\$1 > 0.0))$. The table in Figure 3 summarizes the results obtained from symbolically executing P2. The path condition for P2 is simply the conjunction of the branch conditions B1 through B11. One path constraint

Partial Path	Last Edge Selected	Branch Condition Generated	Partial Path Condition
P2\2+	(2,3)	B1: (N$1 > 0)	B1
P2\6+	(6,7)	B2: (A$1 > 0.0) or (-A$1 > 0.0)	B1 & B2
P2\7+	(7,8)	B3: (A$1 > 0.0)	B1 & B2 & B3
P2\9+	(9,11)	B4: (-A$1 ≤ A$1)	B1&......&B4
P2\11+	(12,5)	B5: (2 ≤ N$1)	B1&......&B5
P2\13+	(6,11)	B6: (A$2 ≤ A$1) and (-A$2 ≤ A$1)	B1&......&B6
P2\15+	(12,5)	B7: (3 ≤ N$1)	B1&......&B7
P2\17+	(6,7)	B8: (A$3 > A$1) or (-A$3 > A$1)	B1&......&B8
P2\18+	(7,9)	B9: (A$3 ≤ A$1)	B1&......&B9
P2\19+	(9,10)	B10:(-A$3 > A$1)	B1&......&B10
P2\22+	(12,13)	B11:(4 > N$1)	B1&......&B11

P2=(1 2 3 4 5 6 7 8 9 11 12 5 6 11 12 5 6 7 9 10 11 12 13 14)

Figure 3

The Successive Partial Path Conditions That Are
Generated During Symbolic Execution of P2

is defined by each of the branch conditions except B6, which defines two path constraints. Thus, symbolic execution of P2 generates a total of 12 path constraints. The constraints obtained from B2 and B8 are OR-groups and consist of two OR-clauses each, while the remaining ten constraints are AND-constraints. Any assignment of input values to the symbolic names, A$1, A$2, A$3, and N$1, that satisfies the ten AND-constraints and one OR-clause from each of the OR-groups forces execution down the path P2. For example, when (A$1 = 1), (A$2 = 1), (A$3 = -2) and (N$1 = 3), P2 is executed. Using this assignment of values to symbolic names, all of the AND-constraints, the first OR-clause of the first OR-group and the second OR-clause of the second OR-group are satisfied.

CONMAN maintains the AND-constraints and the OR-groups in separate data structures, called the AND-structure and the OR-structure, respectively. Conceptually, the representation of the equalities and inequalities in the two structures is very similar. However, at the implementation level the representations differ for two reasons. First, differences arise from the need for additional information in the OR-structure to maintain the proper partitioning of the OR-clauses into OR-groups. Second, the AND-structure is passed to the inequality solver to determine a data set that exercises the path. Since the AND-structure is a component of the BNB-structure, the data structure that defines the problem to be solved by the inequality solver, it is maintained in the form that is required by the inequality solver (see Appendix A). The next two subsections describe the AND-structure and the OR-structure in more detail. Examples throughout the remainder of this section use the constraints that are generated by symbolic execution of the path P2.

2.2. Conceptual Representation

In this subsection a convenient conceptualization of the constraint structure is introduced. Although not implemented, this abstraction is easier to understand and manipulate than the representation that is actually used. Furthermore, the actual representation is best described in terms of this conceptual representation.

CONMAN puts each linear equality or inequality in a standard form in which the lefthand side (LHS) is written as a linear combination of symbolic names, the righthand side (RHS) as a constant, and the relation between the LHS and the RHS as either less than (<), less than or equal ($\leq$), or equal (=). For example, the constraint (-A\$3 > A\$1) that is generated by B10 is expressed as (A\$1 + A\$3 < 0). It is this standard form of an equality or inequality that is represented in the conceptualization of the constraint structure.

With suitable conventions to determine the association between coefficients and symbolic names, the standard form of a linear equality or inequality can be represented by a vector of coefficients, the constant term and the relation between the LHS and the RHS. This suggests a natural representation for a group of equalities and inequalities using matrices and vectors. Thus conceptually, the AND-structure is composed of three components: a matrix, called the A matrix, in which the coefficients of the AND-constraints are recorded; a vector, called the B vector, that records the constants that appear on the RHS of the AND-constraints; and a vector, called the S vector, that records the relations involved in the AND-constraints. Each row of the A matrix records the vector of coefficients defined by the LHS of one AND-constraint, while the same row (or entry) in the B and S vectors record, respectively, the constant term and the relation that appears in the constraint. As AND-constraints are generated along the path, they are assigned to rows sequentially. Each column of the A matrix corresponds to one symbolic name. Thus, the symbolic name that is associated with a coefficient determines the column of the A matrix in which the coefficient is recorded. The ATTEST system assumes a maximum of 100 symbolic names in each program path. The A matrix, therefore, contains 100 columns. Symbolic names that are required to take on integer values are assigned to the first fifty columns in the order of their first appearance in the path constraints. Similarly, symbolic names that are required to take on real values are assigned to the last fifty columns in the order of their first appearance in the path constraints. Figure 4 demonstrates these data structures using the AND-constraints that are generated by symbolically executing P2.

$$\begin{array}{lcccccccccl}
 & \multicolumn{6}{c}{\text{A matrix}} & \text{S vector} & \text{B vector} & & \text{AND-constraint (standard form)} \\
\text{Column number} & 1 & 2\ldots50 & 51 & 52 & 53 & 54\ldots100 & & & & \\
 & -1 & 0\ldots0 & 0 & 0 & 0 & 0\ldots.0 & < & 0 & & -N\$1 < 0 \\
 & 0 & 0\ldots0 & -1 & 0 & 0 & 0\ldots.0 & < & 0 & & -A\$1 < 0 \\
 & 0 & 0\ldots0 & -2 & 0 & 0 & 0\ldots.0 & \leq & 0.0 & & -2A\$1 \leq 0.0 \\
 & -1 & 0\ldots0 & 0 & 0 & 0 & 0\ldots.0 & \leq & -2 & & -N\$1 \leq -2 \\
 & 0 & 0\ldots0 & -1 & 1 & 0 & 0\ldots.0 & \leq & 0.0 & & -A\$1+A\$2 \leq 0.0 \\
 & 0 & 0\ldots0 & -1 & -1 & 0 & 0\ldots.0 & \leq & 0.0 & & -A\$1-A\$2 \leq 0.0 \\
 & -1 & 0\ldots0 & 0 & 0 & 0 & 0\ldots.0 & \leq & -3 & & -N\$1 \leq -3 \\
 & 0 & 0\ldots0 & -1 & 0 & 1 & 0\ldots.0 & \leq & 0.0 & & -A\$1+A\$3 \leq 0.0 \\
 & 0 & 0\ldots0 & 1 & 0 & 1 & 0\ldots.0 & < & 0 & & A\$1+A\$3 < 0 \\
 & 1 & 0\ldots0 & 0 & 0 & 0 & 0\ldots.0 & < & 4 & & N\$1 < 4
\end{array}$$

Figure 4

Matrix Conceptualization of the AND-structure
After Symbolic Execution of P2

The conceptualization of the OR-structure parallels the matrix representation of the AND-structure described above. Like the AND-structure, the OR-structure can be envisioned as consisting of three components: a matrix of coefficients, called the OR matrix; a vector for the constant terms, called ORB; and a vector for the relations, called ORS. An entry in ORB and ORS and a row of the OR matrix are associated to an OR-clause in the order in which the clause is encountered along the path. The OR matrix, like the A matrix, contains one hundred columns. The same association between symbolic names and columns is maintained in the OR-matrix as is maintained in the A matrix. Corresponding columns of the A matrix and the OR matrix are thereby associated to the same symbolic names. To delimit individual OR-groups, the OR-structure maintains an additional vector, called ORGRP. The entries of ORGRP identify the rows of the OR matrix that contain the first clause of each OR-group. The OR matrix and the ORB and ORS vectors record the actual equalities and inequalities involved in the OR-clauses, while ORGRP maintains the proper partitioning of the OR-clauses into OR-groups. Figure 5 demonstrates these data structures using the OR-groups that are generated by symbolically executing P2.

At the implementation level, a sparse array is used to represent the matrices in the OR structure and the AND structure. Since the AND-structure is a component of the BNB structure, however, an additional transformation is required before AND constraints can be put into this sparse

$$
\begin{array}{lccccccc}
 & & & \text{OR-matrix} & & & & \\
\text{Column Number} & 1 & 2..50 & 51 & 52 & 53 & 54..100 \\
 & 0 & 0...0 & -1 & 0 & 0 & 0...0 \\
 & 0 & 0...0 & 1 & 0 & 0 & 0...0 \\
 & 0 & 0...0 & 1 & 0 & -1 & 0...0 \\
 & 0 & 0...0 & 1 & 0 & 1 & 0...0
\end{array}
\quad
\begin{array}{c}
\text{ORS} \\
\begin{pmatrix} < \\ < \\ < \\ < \end{pmatrix}
\end{array}
\quad
\begin{array}{c}
\text{ORB} \\
\begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}
\end{array}
\quad
\begin{array}{c}
\text{OR-clause} \\
\text{(standard form)} \\
-A\$1 < 0.0 \\
A\$1 < 0.0 \\
A\$1-A\$3 < 0.0 \\
A\$1+A\$3 < 0.0
\end{array}
$$

$$
\text{ORGRP} \quad \begin{array}{|c|c|c|c}
\hline
1 & 3 & 5 & \\
\hline
\end{array}
$$

Figure 5

Matrix Conceptualization of the OR-structure
After Symbolic Execution of P2

array format. This transformation is described in the next subsection.

2.3. Constraint Transformation

To conform with the standards of the inequality solver, CONMAN is required to transform each AND-constraint. There are two characteristics of the linear inequality solver that make such a transformation necessary. First, it cannot handle strict inequalities; and second, it constrains the values of its variables to be non-negative. Therefore, the transformation of an AND-constraint is a two-step process. The first step, which is omitted for constraints that are not strict inequalities, addresses the first limitation. The second step, which is done for every constraint, addresses the second.

If the relation between the LHS and the RHS of a constraint that is in standard form is $<$, then the first step of this transformation consists of subtracting a small positive constant from the RHS of the constraint and changing the relation to $\leq$. For expositional purposes, $\underline{e}$ is used to denote this small, positive constant. Thus, (A$1 + A$3 < 0), which is the standard form for the constraint (-A$3 > A$1) generated by B10, is changed into (A$1 + A$3 $\leq$ -$\underline{e}$). If the relation between the LHS and the RHS of a constraint in standard form is $\leq$ or =, this first step is omitted.

Since the inequality solver constrains all variables to have non-negative values, the second step in transforming an AND-constraint consists of expressing each symbolic name as the difference between two non-negative valued variables and substituting these expressions into the constraint in place of the original symbolic names. For example, consider (A$1 + A$3 $\leq$ -$\underline{e}$), which is the partially transformed form of the constraint (-A$3 > A$1) mentioned above. The expressions (A$1 = A$1' - R) and (A$3 = A$3' - R) are used to transform this constraint into (A$1' + A$3' - 2R $\leq$ -$\underline{e}$), with A$1', A$2', and R all constrained to be non-negative. Note that the expressions that are substituted for the original symbolic names can be used to translate any solution for the transformed constraint into a solution for the originial constraint. For example, the solution (A$1' = 3), (A$3' = 0), and (R = 2) for

(A$1' + A$3' - 2R $\leq$ -$\underline{e}$)

can be translated to give (A$1 = 1) and (A$3 = -2), a solution for (-A$3 > A$1). Thus for the second step of the transformation, a constraint is expressed in terms of non-negative valued variables. One non-negative valued variable is associated to each symbolic name and, for expositional purposes, is marked by a prime ('). Two additional non-negative valued variables are used to supplement the primed variables. One, the <u>integer translation variable</u>, serves as the subtrahend in expressions for symbolic names that are required to take on integer values. The other, the <u>real translation variable</u>, serves as the subtrahend in expressions for symbolic names that are required to take on real values. The integer and real translation variables are denoted by I and R, respectively. The integer translation variable and the primed variables associated with integer valued symbolic names are required to take on non-negative, integer values, while the real translation variable and the primed variables associated to real valued symbolic names are required to take on non-negative, real values.

When the second step of the transformation is applied to the path P2, the following substitutions are made for the symbolic names.

N$1 = N$1'-I

A$1 = A$1'-R

A$2 = A$2'-R

A$3 = A$3'-R

Figure 6 gives the result of transforming each of the AND-constraints involved in this path. The solution, (A$1' = 3), (A$2' = 1), (A$3' = 0), (N$1' = 3), (R = 2) and (I = 0), for the transformed constraints in this example, would result in (A$1 = 1), (A$2 = -1), (A$3 = -2) and (N$1 = 3) being generated to solve the original constraints.

Conceptually, the transformed constraints can be represented by a matrix of coefficients and by vectors for the relations and constant terms in the same way that the original constraints are. Thus, the S and B vectors represent, respectively, the relations and the constant terms in the transformed constraints. Since the transformed

AND-constraints

Standard form	Transformed form
-N$1 < 0	-N$1'+I ≤ -e
-A$1 < 0	-A$1'+R ≤ -e
-2A$1 ≤ 0.0	-2A$1'+2R ≤ 0.0
-N$1 ≤ -2	-N$1+I ≤ -2
-A$1+A$2 ≤ 0.0	-A$1'+A$2' ≤ 0.0
-A$1-A$2 ≤ 0.0	-A$1'-A$2'+2R ≤ 0.0
-N$1 ≤ -3	-N$1'+I ≤ -3
-A$1+A$3 ≤ 0.0	-A$1'+A$3' ≤ 0.0
A$1+A$3 < 0.0	A$1'+A$3'-2R ≤ -e
N$1 < 4	N$1'-I ≤ 4-e

Figure 6

The AND-constraints Generated by P2, Shown in Standard Form and After Being Transformed by CONMAN constraints involve a primed variable for each of the symbolic names and may involve the integer translation variable or the real translation variable, or both, the matrix of coefficients for the transformed constraints contains 102 columns. The 101st column corresponds to the integer translation variable, I, and the 102nd column corresponds to the real translation variable, R. The first 100 columns correspond to the primed variables in a natural way. The same column that is used for a symbolic name in the A matrix, is used for the associated primed variable in the matrix of coefficients for the transformed constraints. Therefore, since the coefficient of a symbolic name in a constraint and the coefficient of the associated primed variable in the transformed constraint are identical, the A matrix is embedded in the matrix of coefficients for the transformed constraints. It occupies the first one hundred columns of that matrix.

2.4. Sparse Array Representation

For most purposes the AND structure can be envisioned as consisting of the A matrix and the S and B vectors, while the OR structure consists of the OR matrix and the ORS and ORB vectors. In actuality, however, the AND structure consists of the S and B vectors, and of a sparse array representation of the matrix of coefficients for the transformed AND-constraints, while the OR structure consists of the ORS and ORB vectors, and of a sparse array representation of the OR matrix. This subsection describes the sparse array representations.

The sparse array representation that CONMAN uses for the matrix of coefficients for the transformed AND-constraints conforms to the standards of the inequality solver. The non-zero elements of the matrix are stored row by row in a one-dimensional array, called the AA array. A second array, called JCOL, records the number of the column in the coefficient matrix of each entry of the AA array. Successive entries of a third array, called IROW, record the index of the entry in the AA array that contains the first coefficient of successive rows of the coefficient matrix. Thus, the entries of IROW point into the AA array to the beginning of the rows of the coefficient matrix. The number of AND-constraints in the AND-structure is specified by the value of NEWAN. Figure 7 provides a conceptualization of this data structure. Figure 8 shows the sparse array representation for the matrix of coefficients for the transformed AND-constraints that would be generated by

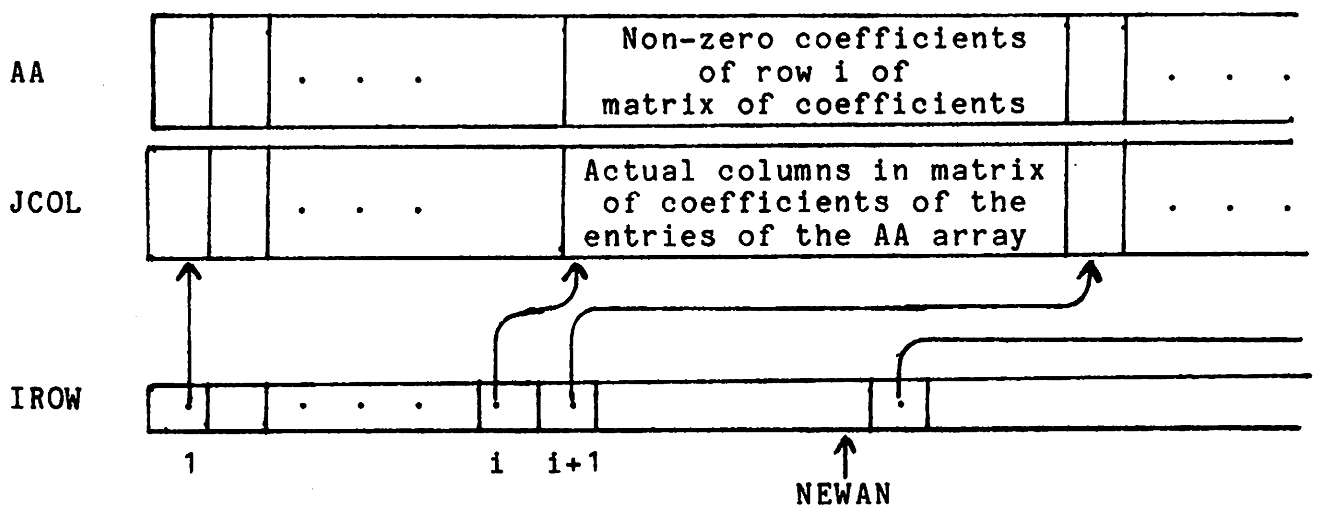

AA
Non-zero coefficients of row i of matrix of coefficients
. . .
JCOL
Actual columns in matrix of coefficients of the entries of the AA array
IROW
1
i
i+1
NEWAN

Figure 7

Conceptualization of the Sparse Array Representation Used in the AND-structure

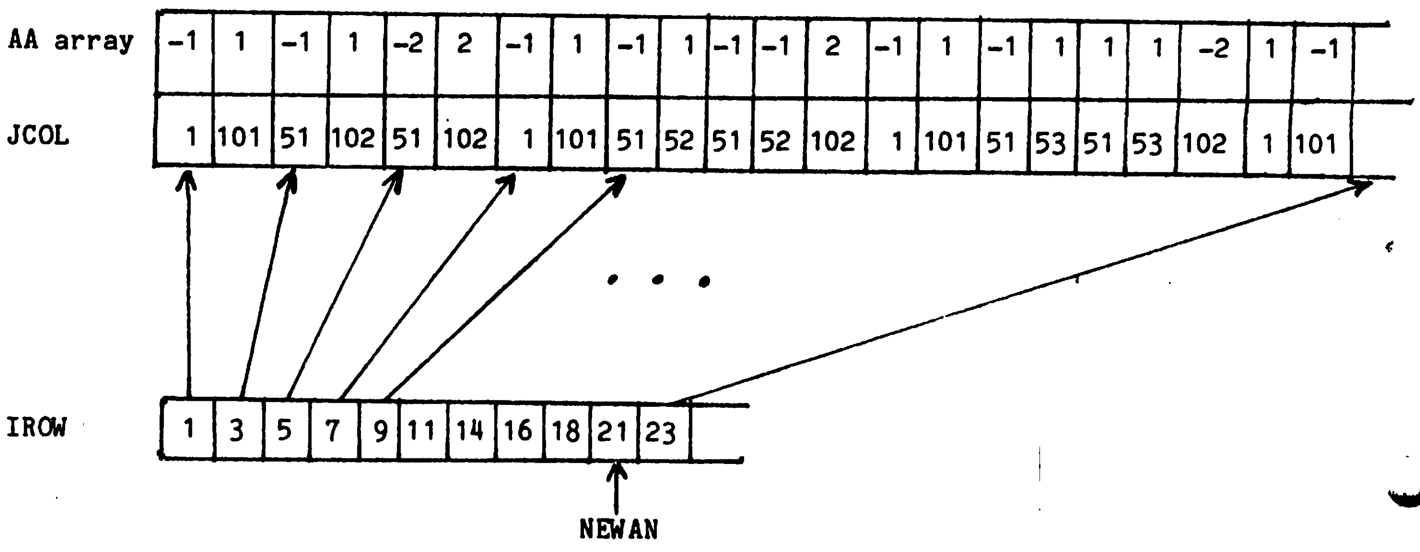


Figure 8

Sparse Array Representation of the Matrix of Coefficients of the Transformed AND-constraints of Figure 6 symbolic execution of P2. Note that the coefficient matrix is completely characterized by the AA, JCOL, and IROW arrays.

The OR matrix is represented in the OR-structure as a sparse array in the same manner that the matrix of coefficients for the transformed constraints is represented in the AND-structure. The non-zero elements of the OR matrix are stored in the ORAA array, row by row. ORCOL signifies to which column of the OR matrix each entry of the ORAA array belongs, and ORROW signifies in which element of the ORAA array each row of the OR matrix begins. The value of NEWOR indicates the number of OR-groups that are recorded in the OR structure. Figure 9 shows the sparse array representation of the OR matrix of Figure 5. Note that the OR clauses have not had the two translation steps performed on them. This is because the BNB structure, which defines the problem to be solved by the inequality solver, does not use the OR-structure. When a solution is required to satisfy a particular OR-clause, CONMAN translates the OR-clause before representing it in the BNB structure (see Appendix A).

This section has described how the constraints that are generated when a program path is symbolically executed are represented in the constraint structure. CONMAN does not record these constraints in one indivisible step. Instead, it begins with an empty constraint structure and, as long as the partial path condition is consistent, it enters each constraint as it is generated during the symbolic execution

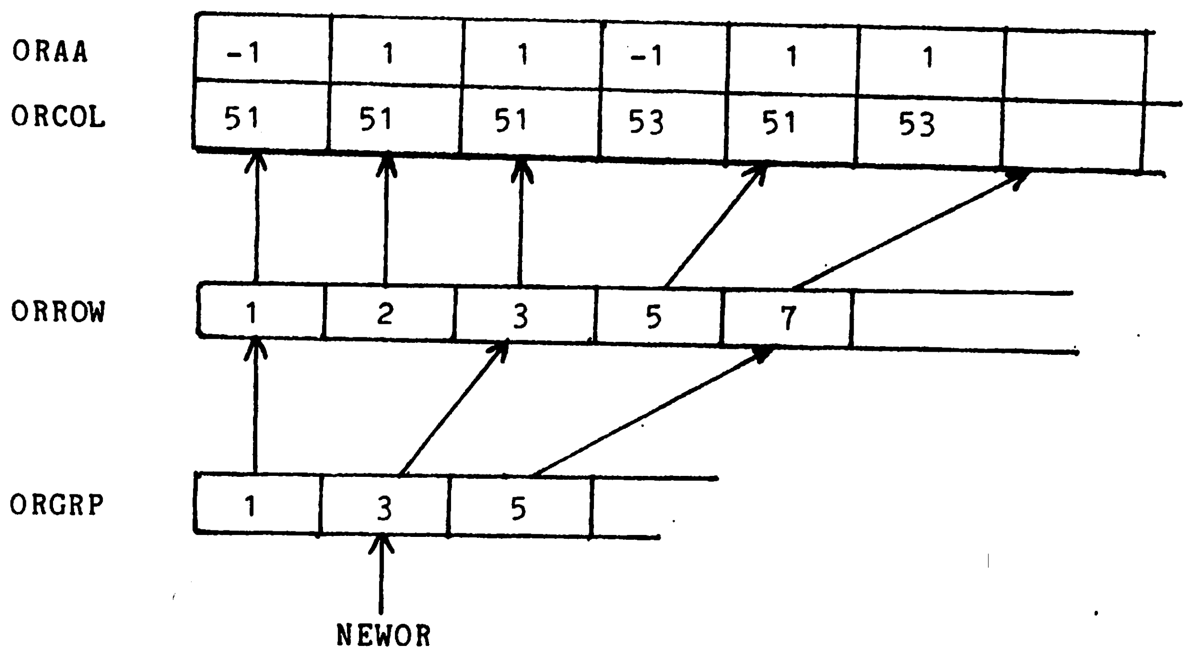


Figure 9

Sparse Array Representation of the OR-matrix of Figure 5 process. Therefore, at any point during this process the constraint structure represents the partial path condition of that portion of the path that has been symbolically executed. The next section describes the process by which the partial path condition is maintained.

3. Maintaining the Partial Path Condition

CONMAN builds the path condition of a specified program path in an incremental fashion. This section describes the algorithm it uses and develops some terminology that is needed later in this report. Two examples are presented. One illustrates the manner in which the path condition for P2 is constructed. The other demonstrates the steps involved in detecting an infeasible path.

As ATTEST sequentially executes the nodes and edges in a path, CONMAN maintains the constraint structure to represent the partial path condition for that portion of the path that has been executed. The partial path condition is the conjunction of the branch conditions that have been generated so far. Each time ATTEST symbolically executes an edge that is annotated with a branch predicate, CONMAN determines if the branch condition generated by symbolically executing that branch predicate is consistent with the partial path condition that is currently in the constraint structure and updates the constraint structure accordingly. CONMAN accomplishes this in three steps. First, it adds the constraints in the branch condition to the constraint structure. Second, it determines if this new constraint structure is consistent. And third, it updates the constraint structure according to the results obtained in the second step. Each of these steps is described in more detail below.

To simplify the description, assume the following situation. During symbolic execution of a program path, Pk, the partial path, Pk\n, has been executed and the extension of this partial path to Pk\n+ generates a branch condition. At this point, the constraint structure contains the partial path condition for Pk\n, which is referred to as the <u>existing partial path</u>. This condition is called the <u>existing condition</u>. The AND-constraints involved in the existing condition are called <u>old AND-constraints</u>, while the OR-groups are called <u>old OR-groups</u>. Collectively, all of the constraints in the existing condition are called the <u>old constraints</u>. CONMAN maintains two pointers, OLDAN and OLDOR, which point to the last old AND-constaint in the AND structure and the last old OR-group in the OR-structure, respectively. NEWAN and NEWOR, as described in the previous section, point to the last AND-constraint and last OR-constraint in the constraint structure. After symbolically executing Pk\n the old constraints are the only constraints in the constraint structure so that the values of OLDAN and OLDOR coincide with the values of NEWAN and NEWOR.

After symbolically executing the nth edge, CONMAN adds the new branch condition to the constraint structure. The constraints that define this branch condition are called the <u>new constraints</u>. CONMAN adds <u>new AND-constraints</u> to the AND-structure directly below the old AND-constraints and <u>new OR-groups</u> to the OR-structure directly below the old OR-groups. After this addition OLDAN and NEWAN delimit the old and new sections of the AND structure, while OLDOR and NEWOR delimit the old and new sections of the OR-structure. The division of the constraint structure into new and old sections after the application of this first step is depicted in Figure 10. The conjunction of all the constraints in the constraint structure, called the <u>augmented condition</u>, defines the partial path condition for Pk\n+.

After creating the augmented condition, CONMAN analyzes it to determine if it is consistent. A detailed description of how CONMAN analyzes the augmented condition is deferred to the next section. The partitioning of the constraint structure into old and new sections is maintained during this analysis. The constraint structure is then updated according to the results obtained in the analysis step. If the augmented condition is consistent, the new branch condition is incorporporated into the existing condition by repositioning the pointers OLDAN and OLDOR to NEWAN and NEWOR, respectively. Thus, the constraint structure is updated to represent the partial path condition for Pk\n+ and symbolic execution proceeds along the program path Pk. If the augmented condition is inconsistent, the branch condition is removed from the constraint structure by repositioning NEWAN and NEWOR to OLDAN and OLDOR, respectively. Thus, the constraint structure is returned to representing the partial path condition for Pk\n. In this later case, the original path, Pk, is infeasible. Symbolic execution of Pk must be abandoned, but a different path can

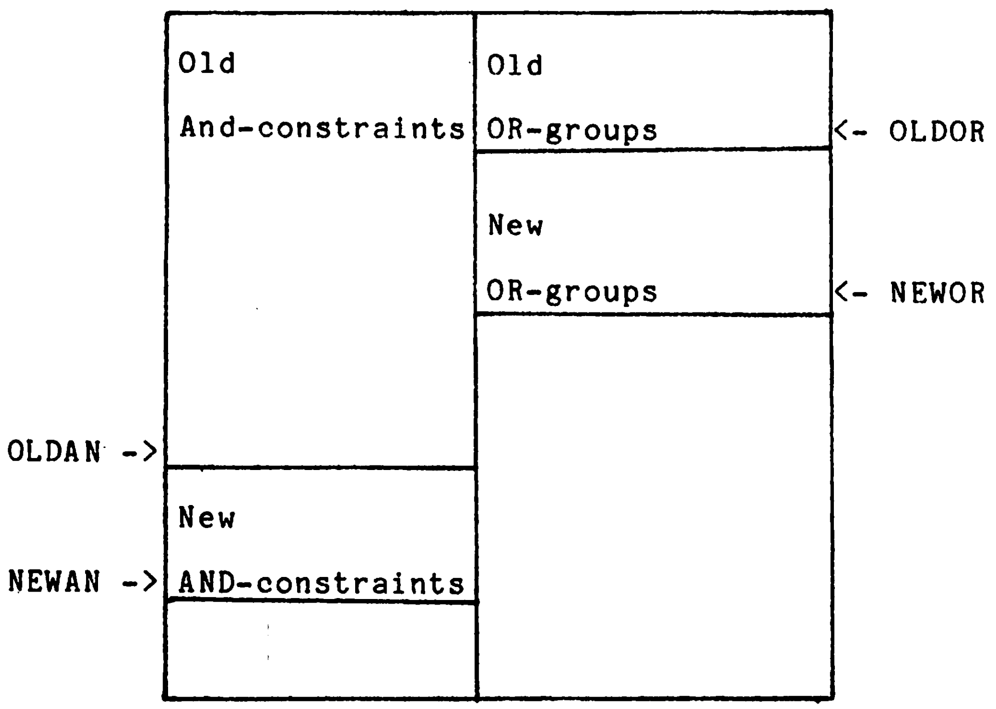


Figure 10

Partitioning of the Constraint Structure into New and Old Sections be symbolically executed, beginning at its nth edge. The new path must extend Pk\n by selecting an edge that is not associated with the offending branch predicate.

The process of maintaining the path condition is illustrated using the path P2. The diagram in Figure 11 shows the constraint structure after each of the first four branch predicates have been encountered during symbolic execution of this path and before each augmented condition has been analyzed. Initially the constraint structure, which is empty, represents the constant logical condition TRUE. As this is the partial path condition for P2\2, the old section of the constraint structure represents the partial path condition for P2\2, which is the existing partial path prior to analysis of P2\2+. The constraint (N$1 > 0) defines the branch condition generated by symbolic execution of the edge (2,3), the first annotated edge in P2. Thus, this is the only constraint in the new section of the constraint structure in the first diagram, and the augmented condition is the partial path condition for P2\2+. Since this augmented condition is consistent, the above branch condition is part of the existing condition in the diagram for P2\6+. The new section of the constraint structure in this diagram consists of the single OR-group, ((A$1 > 0.0) or (-A$1 > 0.0)), which defines the branch condition generated by symbolic execution of the next annotated edge along the path. The old section represents the partial path condition for P2\6. The augmented condition is the partial path condition for P2\6+. Again, the augmented condition is

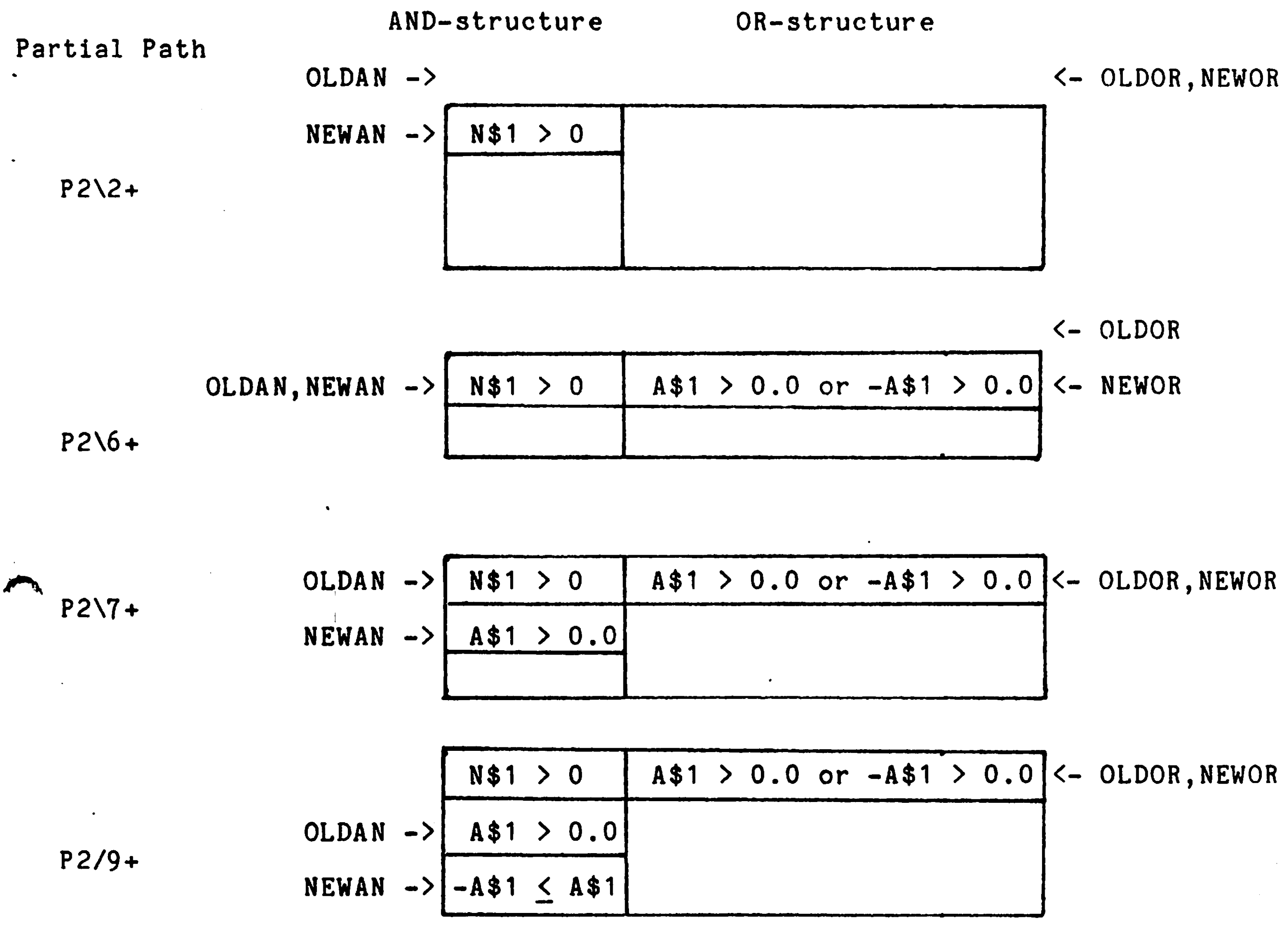


Figure 11

Evolution of the Constraint Structure as the First Four Branch Predicates are Encountered during Symbolic Execution of P2 consistent. Similarly, the diagrams for P2\7+ and P2\9+ show the constraint structure after the next two branch predicates have been encountered and before the structure has been updated.

For another example, consider the infeasible path P3=(1 2 3 4 5 6 7 9 11 12 13 14). The top diagram of Figure 12 depicts the constraint structure after the fourth branch condition has been recorded in the new section of the constraint structure. At this point the augmented condition, which is the partial path condition for P3\8+, is inconsistent. Therefore, as shown in the bottom diagram of the same figure, the fourth branch condition is deleted from the constraint structure after analysis of this augmented condition. Symbolic execution of the original path, P3, must be abandoned, but symbolic execution may continue along any path that extends P3\8 by selecting the edge (9,10). For example, the path P4=(1 2 3 4 5 6 7 9 10 11 12 13 14) may be symbolically executed using the results already obtained from analysis of P3\8, which coincides with P4\8.

Thus, CONMAN builds the path condition in stages that correspond to the different partial path conditions that are generated as the path is symbolically executed. This section has described how the transition between stages is made.

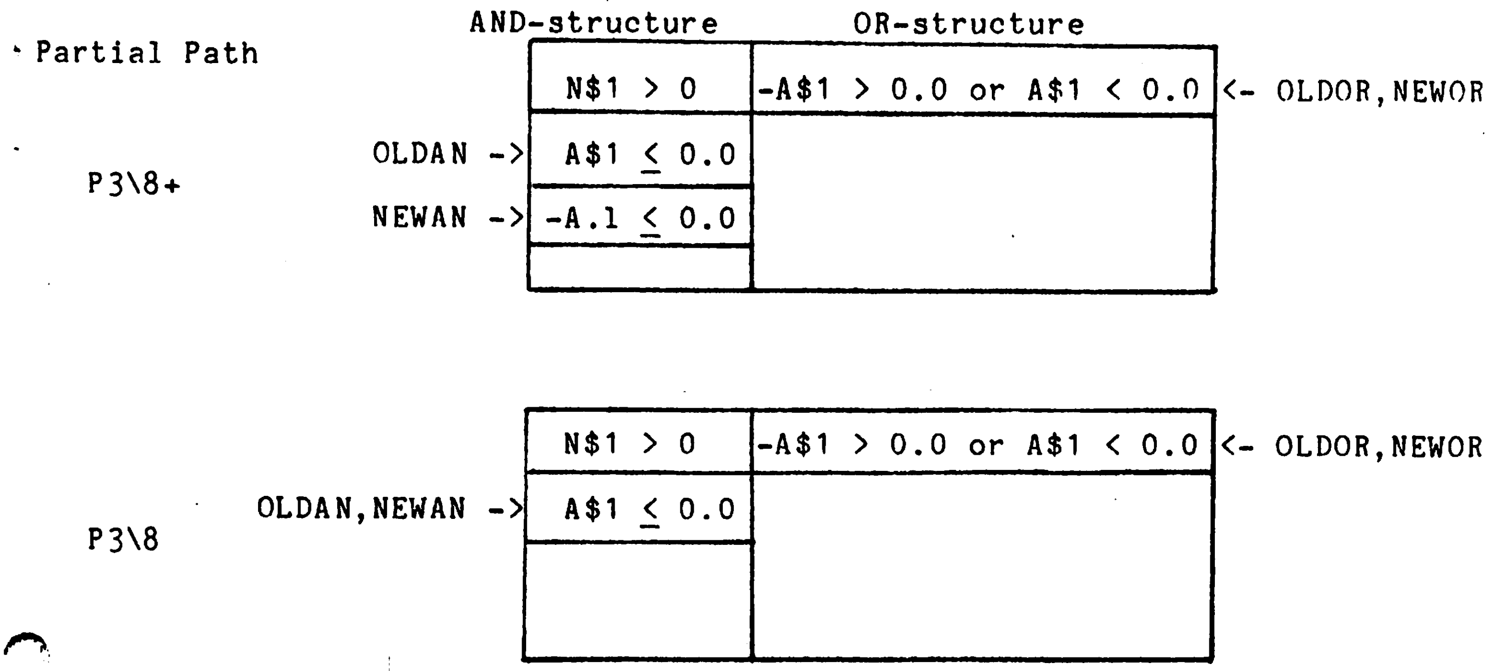


Figure 12

Constraint Structure Immediately Before (top) and After (bottom) Detection of an Infeasible Path

4. Analysis of the Partial Path Condition

The previous section describes the method by which CONMAN maintains the partial path condition. This section discusses how the partial path is analyzed. The routine responsible for this analysis is called TRYPTH. A high level overview of the algorithm used in TRYPTH is developed in the first subsection of this section. First, the terminology needed for this discussion is presented. Then, the overview of TRYPTH is developed and a convention that simplifies this algorithm is discussed. The detailed description of TRYPTH that is presented in the second subsection assumes a thorough understanding of this convention and of the implications that can be drawn from it. This description is followed by an example that uses the path P2.

4.1 Overview of TRYPTH

As described in the previous section, CONMAN represents the augmented condition in the constraint structure before it calls TRYPTH. TRYPTH then attempts to generate a set of values that, if assigned to the symbolic names in the path, would satisfy the augmented condition. If it succeeds, the augmented condition is consistent and a data set that would exercise the augmented partial path has been generated. If it can show that there is no assignment of values to symbolic names that satisfies the augmented condition, the augmented condition is inconsistent. In this manner TRYPTH simultaneously determines if the augmented partial path is consistent and if it is, generates test data. The terminology that is developed below facilitates a more precise statement of what is involved in the analysis as it has been outlined above.

Any assignment of values to symbolic names that satisfies the augmented condition must satisfy all of the AND-constraints and at least one OR-clause from each OR-group in the augmented condition. To generate such an assignment, TRYPTH must first select an OR-clause from each OR-group so that the conjunction of the selected OR-clauses and the AND-constraints is satisfiable. An <u>OR-clause selection</u> refers to a choice of one OR-clause from each of the first K OR-groups, where NEWOR $\geq$ K $\geq$ 0. TRYPTH uses a vector, called a <u>configuration</u>, to represent a given OR-clause selection. The OR-groups along the path are associated to the entries of the configuration in the order in which the OR-groups are recorded in the OR structure. Each of the first K′ entries records the number of the OR-clause that the OR-clause selection chooses from the associated OR-group. All other entries are zero, indicating that the OR-clause selection does not select an OR-clause from those OR-groups. TRYPTH uses a variable called <u>CONF</u>, for recording configurations. Since CONMAN assumes a maximum of 80 OR-groups in a program path, CONF is an 80-dimensional array. A zero is recorded in each entry of CONF for which there is no OR-group in the augmented condition. In the process of analyzing the augmented condition the value of CONF is continually modified. Its value at the beginning of any step in this process is said to be the starting configuration for that step.

Given a configuration, a second configuration is said to extend the first configuration if the set of OR-clauses selected by the first configuration is contained in the set of OR-clauses selected by the second. A configuration is said to be full if it selects an OR-clause from each OR-group in the augmented condition. A configuration and the OR-clause selection that it represents* are said to be consistent if the conjunction of the AND-constraints and the OR-clauses selected by the configuration is satisfiable. Thus, if a configuration is consistent, values can be assigned to the symbolic names in the augmented condition so that each AND-constraint and each OR-clause selected by the configuration is satisfied. Such an assignment is said to solve the conjunction of the AND-constraints and the OR-clauses selected by the configuration. More simply, it is called a solution for the configuration. TRYPTH maintains a vector, called CFSOL, to represent a solution for the configuration recorded in CONF. CFSOL is a 100-dimensional array in which the values assigned to the symbolic names by the solution are recorded. The column in

---

*Here and in the future, the OR-clause selection that a configuration represents and the configuration itself, are used interchangeably. This avoids terminology such as "the OR-clauses chosen by the OR-clause selection that is represented by the configuration", which is more simply stated as "the OR-clauses selected by the configuration".

the A array that corresponds to a symbolic name determines the column (entry) of CFSOL that corresponds to the same symbolic name. By convention, symbolic names that are not involved in the augmented condition are assigned a zero by any solution. Thus, any entries of CFSOL for which there are no symbolic names are zero. As the value of CONF is modified by TRYPTH, the value of CFSOL is updated accordingly. The value of CFSOL at the beginning of any step in TRYPTH is said to be the <u>starting solution</u> for that step. Using the terminology developed above, the augmented condition is consistent if there exists a full consistent configuration -- that is, if there exists a configuration whose first NEWOR entries are all non-zero and for which there is a solution.

For example, the configuration [1, 0] selects the first OR-clause from the first OR-group in the path condition for P2. When P2\6+ is being analyzed by TRYPTH, this is a full, consistent configuration (see Figure 3). A solution for this configuration is given by (N$1 = 1), (A$1 = 1), (A$2 = 0) and (A$3 = 0). Thus, the partial path condition for P2\6+ is consistent. Although [1, 0] is a consistent configuration during analysis of P2\19+, it is not a full configuration since it doesn't select an OR-clause from the second OR-group in the augmented condition. The configuration [1, 1] extends the above configuration to a full configuration. This configuration is not consistent, however, since the OR-clause that it selects from the second OR-group, (A$3 > A$1), is inconsistent with (-A$3 > A$1), the AND-constraint that was generated by symbolic execution of the edge (9,10). The partial path condition for P2\19+ is consistent though, since [1, 2] is a full, consistent configuration. The assignment (N$1 = 3), (A$1 = 1), (A$2 = 0) and (A$3 = -2) is a solution for this configuration.

The inequality solver can be used to test a configuration for consistency. TRYPTH can invoke the inequality solver to generate a solution for the set of equalities and inequalities defined by the AND-constraints and the OR-clauses that are selected by a particular configuration (see Appendix A). If a solution is returned, the configuration is consistent. Otherwise, the configuration is inconsistent.

It is sometimes more efficient, however, to use the results obtained from analyzing an earlier partial path to determine if a particular configuration is consistent. This is the case, for example, if the new condition consists only of AND-constraints and if the new AND-constraints are implied by old constraints. Then the configuration and the solution that were generated when the existing condition was analyzed in an earlier call to TRYPTH represent a full, consistent configuration and solution for the augmented partial path. The results of analyzing an earlier partial path can also be used to conclude that certain configurations are inconsistent. In particular, a configuration is inconsistent if it extends a configuration that was found to be inconsistent in an earlier call to

TRYPTH. It is more efficient to use the results obtained from the analysis of the earlier partial path than it is to use the inequality solver in situations such as those described above.

To avoid unnecessary calls to the inequality solver, the configuration and the solution that were obtained when the existing condition was analyzed by TRYPTH are made available during the analysis of an augmented condition. These are referred to as the <u>existing configuration</u> and the <u>existing solution</u> and are recorded in the variables <u>EXCON</u> and <u>EXSOL</u>, respectively. After TRYPTH determines if the augmented condition is consistent, it updates EXCON and EXSOL for the next call to TRYPTH. Thus, if analysis of the augmented condition reveals it to be consistent, TRYPTH records a full, consistent configuration and solution in EXCON and EXSOL. Otherwise, it does not alter their values. At the call to TRYPTH, therefore, EXCON and EXSOL contain the existing configuration and existing solution that are to be used during analysis of the current augmented condition. Note that the existing solution is known to solve the conjunction of the old AND-constraints and the OR-clauses that are selected by the existing configuration. If it also satisfies the new AND-constraints and an OR-clause from each of the new OR-groups, the augmented condition is consistent and the inequality solver does not have to be called.

To avoid unnecessary calls to the inequality solver, TRYPTH also adheres to a convention that permits it to use the existing configuration to identify certain inconsistent configurations. First, the set of all possible configurations for a given path is considered to be ordered using the standard lexicographical order relation. Thus, a configuration is <u>less than</u> those configurations that it precedes lexicographically. Then, if the augmented condition is consistent, TRYPTH generates the smallest, full, consistent configuration and a corresponding solution, rather than an arbitrary full, consistent configuration and solution. Because of this convention, any configuration that is less than the existing configuration extends a configuration that was found to be inconsistent in an earlier call to TRYPTH. Thus, the existing configuration establishes a lower bound on the set of full, consistent configurations. Only configurations larger than the existing configuration are considered by TRYPTH when it analyses the augmented condition.

In its first attempt to find the smallest, full, consistent configuration and a corresponding solution, TRYPTH uses the existing configuration, the existing solution, and any redundancies and inconsistencies that it detects between the new and old constraints. The method for detecting redundancies and inconsistencies is described in Section 5. If this attempt fails, TRYPTH uses the inequality solver to determine if any of the configurations larger than the existing configuration are consistent.

Reconfiguration, the method for finding the next largest consistent configuration is described in Section 6. If the smallest, full, consistent configuration and a corresponding solution are produced, the augmented condition is consistent and the solution provides a data set that would exercise the augmented partial path. Otherwise, the augmented condition is inconsistent.

4.2 Description of TRYPTH

CONF is essentially used as a counter by TRYPTH. When TRYPTH is called, CONF and CFSOL are initialized with EXCON and EXSOL, respectively. Thus, TRYPTH starts with the existing configuration and the existing solution. As it analyzes the augmented condition TRYPTH increments CONF, updating CFSOL appropriately, until either CONF contains the smallest, full, consistent configuration and CFSOL contains a corresponding solution or the augmented condition is determined to be infeasible.

Recall that when TRYPTH is invoked, the branch condition generated by symbolically executing an annotated edge has been recorded in the constraint structure. At this point, the constraint structure is divided into old and new sections. The old section represents the existing condition, while the new section represents the branch condition that has just been generated. If the new section contains AND-constraints TRYPTH invokes the subroutine PROANS to process the new AND-constraints. Then, if the new AND-constraints are found to be consistent with the old

AND-constraints and the old OR-groups and if the new section of the constraint structure contains new OR-groups, TRYPTH invokes the subroutine PROGP to process each new OR-group. PROANS and PROGP are described below.

PROANS analyzes the constraints in the augmented condition as if there were no new OR-groups to satisfy. It starts with the existing configuration and the existing solution and looks for the smallest, consistent configuration that selects an OR-clause from each old OR-group, but does not select an OR-clause from any of the new OR-groups. PROANS' algorithm is summarized in Figure 13. The five steps in this algorithm, which are executed sequentially, are described in more detail below. As soon as a step produces the required configuration or reveals that the augmented condition is inconsistent, PROANS returns, bypassing any subsequent steps.

In the first step, PROANS looks for redundancies and inconsistencies between the new AND-constraints and the old AND-constraints. The types of redundancies and inconsistencies that it detects and the algorithm with which it detects them are described in Section 5. If any redundancies are detected, the constraint structure is simplified accordingly. In the process of simplifying the constraint structure, individual AND-constraints may be killed. The modifications that are made to the constraint structure for this purpose are described in Section 7.

```
(1) Check new AND-constraints and old AND-constraints for
    redundancies
    and inconsistencies;

    If any inconsistencies are detected then
        return (infeasible path);

    If the old AND-constraints imply the new ones then
        return;

(2) Substitute starting solution into each new
    AND-constraint;

    If all are satisfied then
        return;

(3) Call inequality solver to test the starting
    configuration
    for consistency;

    If consistent then
        CFSOL = the solution generated and
        return;

(4) Call inequality solver to test just the
    AND-constraints for consistency;

    If inconsistent then
        return (infeasible path);

(5) Reconfigure (OLDOR specifies the last OR-group to
    include);
```

Figure 13

Summary of PROANS' Algorithm

If a new AND-constraint is found to be inconsistent with an old AND-constraint, or if all of the new AND-constraints are implied by old AND-constraints, PROANS returns in the first step. If the former situation occurs, the augmented condition is inconsistent. If the later situation occurs, PROANS' starting solution, which is already known to satisfy the old AND-constraints and the old OR-clauses selected by PROANS' starting configuration, also satisfies the new AND-constraints. In this situation PROANS kills the new AND-constraints and returns in this step, without altering the values of CONF and CFSOL.

In the second step, PROANS determines whether the starting solution satisfies the new AND-constraints. If it does, PROANS returns in this step without altering the values of CONF and CFSOL.

In the third step, PROANS invokes the inequali;ty solver to generate a solution for the starting configuration. If the inequality solver succeeds in generating a solution, PROANS records the solution in CFSOL and returns in this step without altering the value of CONF.

In the fourth step, PROANS uses the inequality solver to determine if the AND-constraints are consistent by themselves. It returns in this step if they are found to be inconsistent.

The fifth step of PROANS involves a process known as <u>reconfiguration</u>. In general, when provided with a positive integer, N, indicating the number of the last OR-group to be used, the reconfiguration process looks for the smallest, consistent configuration that is greater than its starting configuration and that selects an OR-clause from precisely the first N OR-groups. If it succeeds in this endeavor, it increments CONF to record this configuration and updates CFSOL appropriately. The reconfiguration algorithm is described in more detail in Section 6. In the last step of PROANS, the reconfiguration process starts with PROANS' starting configuration and OLDOR is provided to indicate the last OR-group to be included in the reconfiguration. If one exists, the smallest consistent configuration that selects an OR-clause from each old OR-group and a corresponding solution are recorded in CONF and CFSOL by the reconfiguration process. Otherwise, the augmented condition is inconsistent. In either case, PROANS returns after this step has been executed.

If there are no new OR-groups in the augmented condition or if PROANS finds the augmented condition to be inconsistent, TRYPTH is done. Otherwise, TRYPTH continues the analysis of the augmented condition by processing each new OR-group sequentially, in seperate calls to PROGP. During a call to PROGP, the OR-group that is being processed is referred to as the <u>focal group</u>. Thus, the first new OR-group in the augmented condition is the focal group during the first call to PROGP. As long as the augmented condition is not found to be inconsistent in this call, PROGP is called again with the next new OR-group as the focal group. Continuing in this manner, PROGP is called until either all the new OR-groups are processed or the augmented condition is found to be inconsistent. TRYPTH then returns the results of the last call to PROGP.

PROGP analyzes the constraints in the augmented condition as if only the OR-groups through the focal group needed to be satisfied. Thus, if the focal group is the Mth group in the constraint structure, PROGP looks for the smallest, consistent configuration that selects an OR-clause from precisely the first M OR-groups. If such a configuration exists, PROGP records this configuration in CONF and a solution for it in CFSOL. Otherwise, the augmented condition is inconsistent.

The first time it is called, PROGP starts with the configuration and solution produced by PROANS or, if there are no new AND-constraints in the augmented condition, with the existing configuration and existing solution. Subsequently, PROGP starts with the configuration and solution produced by the previous call to PROGP. Thus, when PROGP is invoked, the conjunction of the AND-constraints with the OR-groups up to, but not including, the focal group is known to be consistent. Furthermore, PROGP's starting configuration and starting solution record, respectively, the smallest, consistent configuration that selects an OR-clause from precisely these OR-groups and a corresponding solution. Since PROGP's starting configuration is less than every consistent configuration that selects an OR-clause from each OR-group through the focal group, it provides a lower bound for the set of configurations that are considered by PROGP.

Like PROANS, PROGP consists of five basic steps. The steps in PROGP, which parallel the corresponding steps in PROANS, are summarized in Figure 14. To facilitate the exposition, the extension of a configuration to the next OR-group has been indicated in this figure by enclosing the configuration that is being extended and the OR-clause that is being selected in square brackets. Thus,

[starting configuration, 2]

denotes the configuration that extends the starting configuration by selecting the second OR-clause from the next OR-group. As in PROANS, each subsequent step in PROGP is executed only if the desired configuration and solution are not generated in a previous step and the augmented condition is not found to be inconsistent. The five steps of PROGP are described in more detail below.

In the first step, the clauses of the focal group are checked for redundancies and inconsistencies with the AND-constraints (again, see Section 5). If any redundancies or inconsistencies are detected in this step, the constraint structure is simplified accordingly. In the process of simplifying the constraint structure, individual OR-clauses of the focal group or the entire focal group may be <u>killed</u>. The circumstances under which OR-clauses and OR-groups are killed and the implementation details are discussed in Section 7.

```
(1) Check OR-clauses of focal group and AND-constraints for
    redundancies and inconsistencies;

    If all OR-clauses are inconsistent with the
        AND-constraints then
        return;

    If an AND-constraint implies an OR-clause then
        kill the focal group and return;

(2) Substitute starting solution into first OR-clause of the
    focal group;

    If the OR-clause is satisfied then
        CONF = [starting configuration,1] and return;

(3) k = number of OR-clauses in the focal group;

    For I = 1, 2,..., k:
        Call the inequality solver to test
            [starting configuration,I] for consistency;

        If consistent then
            CONF = [starting configuration,I],
            CFSOL = solution generated, and
            return;

(4) For I = 1, 2,..., k:
        Call inequality solver to test clause I of the focal
            group
            for consistency with just the AND-constraints;

        If inconsistent then
            kill clause I;

        If consistent then
            go to step (5);

    Return (infeasible path);

(5) Reconfigure (focal group is last OR-group to include);
```

Figure 14

Summary of PROGP's Algorithm

If each OR-clause in the focal group is found to be inconsistent with a constraint in the AND-structure, or if one of the OR-clauses in the focal group is implied by a constraint in the AND-structure, PROGP returns in this step. If the former situation occurs, the augmented condition is inconsistent. If the later situation occurs, PROGP kills the focal group, modifies CONF to reflect the fact that this OR-group has been killed (see Section 7) and returns, without altering CFSOL.

In the second step, PROGP's starting solution is substituted into the first OR-clause in the focal group. If this assignment of values to symbolic names satisfies this OR-clause, PROGP increments CONF so that it extends PROGP's starting configuration to the first OR-clause of the focal group and returns without altering CFSOL.

In the third step, the inequality solver is used to determine if PROGP's starting configuration can be consistently extended to the focal group. The configurations that extend the starting configuration by selecting an OR-clause from the focal group are tested in an ascending sequence. If the inequality solver returns a solution for one of these configurations, the testing sequence is terminated, the configuration and solution are recorded in CONF and CFSOL, and PROGP returns in this step.

In the fourth step, PROGP determines if any of the OR-clauses in the focal group are consistent with just the AND-constraints. If not, there is no need to perform a reconfiguration, for the augmented condition is inconsistent and PROGP returns. If PROGP finds an OR-clause in the focal group that is consistent with the AND-constraints, it kills any OR-clauses that it found to be inconsistent with the AND-constraints and proceeds to the fifth step.

In its final step, PROGP uses the reconfiguration process. The reconfiguration in this step starts with PROGP's starting configuration. The number of the focal group is provided to indicate the the last OR-group to be used in the reconfiguration. If the augmented condition is consistent, the required configuration and a corresponding solution are recorded in CONF and CFSOL by the reconfiguration process. Otherwise, the augmented condition is inconsistent. In either case, PROGP returns at the end of this step.

Analysis of the augmented condition is complete when the new AND-constraints and each new OR-group have been analyzed or when the augmented condition is found to be infeasible. TRYPTH then updates the constraint structure, as described in the previous section, and, if necessary, updates EXCON and EXSOL in preparation for the next call to TRYPTH. If the augmented condition is consistent, it may be able to be simplified further. If so, TRYPTH makes the appropriate modifications to the constraint structure at this time and then returns the solution recorded in CFSOL. If the augmented condition is inconsistent, TRYPTH restores any old constraints that were modified in the first step of PROANS or in the first step of PROGP. The manner in which this is done is described in Section 7.

To illustrate this analysis, consider the path P2. Figure 15 summarizes the results of the eleven calls that are made to TRYPTH during symbolic execution of this path. The smallest, full, consistent configuration and the solution that are returned by each call to TRYPTH have been indicated, along with the step in which they are generated. As there are only two OR-groups in the path condition for P2, only the first two entries of each configuration are displayed. All other entries are zero. The value of the four-tuple (N$1, A$1, A$2, A$3) is used to represent a solution.

Prior to the first call to TRYPTH, CONMAN initializes the existing configuration and existing solution. Since the existing condition has the constant value TRUE in the first call to TRYPTH, the existing configuration is initialized to be the zero vector, while the existing solution is initialized to assign a zero to each symbolic name.

The partial path condition for P2\2+, (N$1 > 0), is analyzed in a single call to PROANS. The first two steps of PROANS fail to generate the required configuration or to detect an inconsistent augmented condition. In the third step the inequality solver returns the solution (N$1 = 1). Thus, PROANS records this solution in CFSOL, with all other symbolic names assumed to be zero, and returns. At this point, analysis of the augmented condition is complete. The augmented condition cannot be simplified. Thus, TRYPTH incorporates the new condition into the existing condition, records the configuration and solution that it generated in

Partial Path analyzed by TRYPTH	New Constraints	Value generated for CONF	Value generated for CFSOL (N$1, A$1, A$2, A$3)	Step in which generated
P2\2+	N$1 > 0	[0,0]	(1, 0.000, 0.000, 0.000)	PROANS,step 3
P2\6+	(A$1 > 0.0) or (-A$1 > 0.0)	[1,0]	(1, 0.001, 0.000, 0.000)	PROGP,step 3
P2\7+	A$1 > 0	[1,0]	(1, 0.001, 0.000, 0.000)	PROANS,step 2
P2\9+	-A$1 ≤ A$1	[1,0]	(1, 0.001, 0.000, 0.000)	PROANS,step 1
P2\11+	2 ≤ N$1	[1,0]	(2, 0.001, 0.000, 0.000)	PROANS,step 3
P2\13+	(A$2 ≤ A$1) and (-A$2 ≤ A$1)	[1,0]	(2, 0.001, 0.000, 0.000)	PROANS,step 2
P2\15+	3 ≤ N$1	[1,0]	(3, 0.001, 0.000, 0.000)	PROANS,step 3
P2\17+	(A$3 > A$1) or (-A$3 > A$1)	[1,1]	(3, 0.001, 0.000, 0.002)	PROGP,step 3
P2\18+	A$3 ≤ A$1	[1,2]	(3, 0.001, -.001, -.002)	PROANS,step 5
P2\19+	-A$3 ≥ A$1	[1,2]	(3, 0.001, -.001, -.002)	PROANS,step 2
P2\22+	4 > N$1	[1,2]	(3, 0.001, -.001, -.002)	PROANS,step 2

Figure 15

Analysis of the Path P2

EXCON and EXSOL, and returns the solution.

The partial path condition for P2\6+ is analyzed in the next call to TRYPTH. For this call, the existing configuration is the zero vector and the existing solution, the solution for P2\2+ given above, is a solution for P2\6. Since the new condition consists of the single OR-group, ((A$1 > 0.0) or (-A$1 > 0.0)), analysis of the augmented condition requires a single call to PROGP. The existing configuration and existing solution provide the starting configuration and starting solution for this call. Since the first two steps of PROGP fail to generate a desired configuration, the configuration [1, 0], which extends PROGP's starting configuration to select the first OR-clause from the first OR-group, is tested for consistency in the third step of PROGP. The inequality solver returns the solution (N$1 = 1) and (A$1 = .001) for this configuration. Thus, PROGP increments CONF to contain this configuration, records this solution in CFSOL, with all other symbolic names assumed to be zero, and returns. After updating the constraint strucuture, EXCON and EXSOL, TRYPTH returns the solution generated by the call to PROGP.

The above configuration and solution define the existing configuration and existing solution when TRYPTH is called to analyze P2\7+. This analysis terminates successfully in the second step of PROANS, when PROANS' starting solution is found to satisfy the single new AND-constraint, (A$1 > 0.0). TRYPTH is invoked to analyze P2\9+, with the same existing configuration and existing solution as before. This time, the single new AND-constraint, (-A$1 ≤ A$1), is found to be implied by the old AND-constraint, (A$1 > 0.0), in the first step of PROANS. Therefore, PROANS returns in this step. TRYPTH then simplifies the augmented condition by eliminating the representation of this new constraint from the constraint structure (see Section 7). Finally, TRYPTH updates EXCON and EXSOL and returns.

The configuration [1, 0] and the solution (N$1 = 2) and (A$1 = .001) are generated by analysis of P2\11+. This solution is obtained in the third step of PROANS when the inequality solver returns a solution for PROANS' starting configuration. The same configuration and solution are returned by TRYPTH during analysis of P2\13+, when PROANS' starting solution is found to satisfy the new AND-constraints in the second step of PROANS. Analysis of P2\15+ terminates successfully in the third step of PROANS when the inequality solver generates the solution (N$1 = 3), (A$1 = .001) and (A$2 = 0.0) for the configuration [1, 0]. When the augmented condition is simplified in this call to TRYPTH, the new constraint, (3 ≤ N$1), completely replaces the old constraint, (2 ≤ N$1), generated at P2\11+ (see Section 7). During analysis of the path, P2\17+, CONF is incremented to [1, 1] in the third step of PROGP. The solution (N$1 = 3), (A$1 = .001), (A$2 = 0.0) and (A$3 = .002) is generated for this configuration by the inequality solver and is recorded in CFSOL in this step.

The analysis of P2\18+ differs from the analysis of earlier partial paths. When PROANS' starting configuration [1, 1], is found to be inconsistent, the fourth step of PROANS is executed. Thus, the inequality solver is used to test the AND-constraints for consistency by themselves. Since they are consistent PROANS proceeds to the fifth step of its algorithm, in which the reconfiguration process produces the smallest, full, consistent configuration, [1, 2], and the solution, (N$1 = 3), (A$1 = .001), (A$2 = -.001), (A$3 = -.002).

Analysis of P2\19+ terminates successfully in the second step of PROANS, when PROANS' starting solution is found to satisfy the single new AND-constraint. Finally, the last call to TRYPTH, which analyzes P2\22+, returns in the second step of PROANS with the solution (N$1 =3), (A$1 = .001), (A$2 = -.001), and (A$3 = -.002) for the configuration [1, 2]. Therefore, the path P2 is feasible and the above assignment of values to symbolic names would result in its execution.

5. Redundancy checking

In the first step of PROANS the new AND-constraints are compared to the old AND-constraints to determine if any of the expressions in the two sets are redundant or inconsistent. Similarly, in the first step of PROGP the OR-clauses of the focal group are compared to all of the AND-constraints to detect redundancies and inconsistencies. In both cases, by repeatedly invoking the subroutine REDCK, each new equality or inequality is compared sequentially to each of the AND-constraints in the appropriate set. This section describes both the kind of redundancies that REDCK detects and the algorithm that it uses.

When REDCK is called it is provided with two expressions. One represents the new equality or inequality. The other represents the equality or inequality against which the new expression is to be compared. To facilitate the discussion these are denoted by In and Io, respectively. In and Io may be redundant, inconsistent, or neither. If redundant, they can be redundant in one of the following three ways: In may imply Io, in which case In is said to <u>dominate</u> Io; In may be implied by Io, in which case In is said to be <u>dominated by</u> Io; or the conjunction of In and Io may be equivalent to the equality that is obtained by replacing $\leq$, the relational operator in Io, by =, in which case In and Io are said to be <u>replaceable by an equality</u>. REDCK, therefore, determines if In and Io are inconsistent or redundant, and if they are redundant, it also determines which of the above three redundancy relationships applies.

REDCK first determines if the coefficient vector of In is a scalar multiple of the coefficient vector of Io. If it is not, In and Io are neither redundant nor inconsistent. If it is, the coefficient vector of In can be obtained by multiplying the coefficient vector of Io by a non-zero scalar, K. In this case REDCK calculates K and four additional values, Rn, Ro, R and SIGNK. Rn denotes the relational operator in the set { =, $\leq$, < } that appears in In, and Ro denotes the relational operator in the set { =, $\leq$ } that appears in Io. R denotes the relational operator in the set { <, >, = } that makes the following statement true:

$$bn \ R \ (K * bo),$$

where bn and bo represent the constant terms of In and Io. Finally, SIGNK denotes the sign of the scalar K. The value of the four-tuple (Rn, Ro, R, SIGNK) is then used to determine whether or not In and Io are inconsistent or redundant. In addition, if they are redundant it determines the type of the redundancy between them. The table in Figure 16 summarizes how relationships between In and Io are deduced from the thirty-six states of this four-tuple.

For example, assume that In represents the inequality, (2A\$1 - A\$2 $\leq$ 1). If Io represents (6A\$1 - 3A\$2 $\leq$ 15), REDCK obtains the four-tuple, ($\leq$,$\leq$, <, +). Thus, In is found to dominate Io. If Io represents (-6A\$1 + 3A\$2 $\leq$ 15), REDCK obtains ($\leq$, $\leq$, >, -), which indicates that In and Io are neither redundant nor inconsistent. Finally, if Io represents (-6A\$1 + 3A\$2 $\leq$ 3), In and Io are replaceable by

(Rn,Ro,R,SIGNK)	Relation
(=,=,=,+)	In is dominated by Io
(=,=,=,-)	In is dominated by Io
(=,=,<,+)	In and Io are inconsistent
(=,=,<,-)	In and Io are inconsistent
(=,=,>,+)	In and Io are inconsistent
(=,=,>,-)	In and Io are inconsistent
(=,≤,=,+)	In dominates Io
(=,≤,=,-)	In dominates Io
(=,≤,<,+)	In dominates Io
(=,≤,<,-)	In and Io are inconsistent
(=,≤,>,+)	In and Io are inconsistent
(=,≤,>,-)	In dominates Io
(≤,=,=,+)	In is dominated by Io
(≤,=,=,-)	In is dominated by Io
(≤,=,<,+)	In and Io are inconsistent
(≤,=,<,-)	In and Io are inconsistent
(≤,=,>,+)	In is dominated by Io
(≤,=,>,-)	In is dominated by Io
(≤,≤,=,+)	In is dominated by Io
(≤,≤,=,-)	In and Io are replaceable by an equality
(≤,≤,<,+)	In dominates Io
(≤,≤,<,-)	In and Io inconsistent
(≤,≤,>,+)	In is dominated by Io
(≤,≤,>,-)	In and Io are not redundant nor inconsistent
(<,=,=,+)	In and Io are inconsistent
(<,=,=,-)	In and Io are inconsistent
(<,=,<,+)	In and Io are inconsistent
(<,=,<,-)	In and Io are inconsistent
(<,=,>,+)	In is dominated by Io
(<,=,>,-)	In is dominated by Io
(<,≤,=,+)	In dominates Io
(<,≤,=,-)	In and Io are inconsistent
(<,≤,<,+)	In dominates Io
(<,≤,<,-)	In and Io are inconsistent
(<,≤,>,+)	In is dominated by Io
(<,≤,>,-)	In and Io are not redundant nor inconsistent

Figure 16

Redundancy Table

an equality. Note that in this final case, the conjunction of In and Io is equivalent to the equality, (-6A$1 + 3A$2 = 3).

6. Reconfiguration

The reconfiguration process, RECON, is invoked in the fifth step of PROANS and the fifth step of PROGP. At its invocation it is provided with a positive integer to indicate the number of OR-clauses to use in the reconfiguration. This integer is denoted by <u>N</u> throughout this section. If there is a consistent configuration that is greater than RECON's starting configuration and that selects an OR-clause from the first N OR-groups, RECON increments CONF to the smallest such configuration and records a solution for it in CFSOL. Otherwise, RECON returns a flag indicating that all such configurations are inconsistent. Essentially, the reconfiguration algorithm consists of alternately incrementing CONF and invoking the inequality solver to test CONF for consistency, until a consistent configuration that selects an OR-clause from each of the first N OR-groups is found or all such configurations are discovered to be inconsistent.

Based on the manner in which the symbolic names are distributed among the equality and inequality expressions in the constraint structure, it is possible to determine that certain configurations are inconsistent without using the inequality solver. For example, assume that there is a single new AND-constraint and that this new constraint is consistent with the existing AND-constraints but is inconsistent with the existing configuration. Thus, the new AND-constraint added some information that made the existing configuration inconsistent. Changes to the configuration that have no affect on the symbolic names in the new AND-constraint will not alter the configuration's inconsistent state. Thus, instead of enumerating CONF over all configurations that are larger than the starting configuration, RECON skips over configurations that are known to be inconsistent. To determine which equality and inequality expressions in the constraint structure can influence which symbolic names, the expressions are partitioned into equivalence classes. Essentially, the equivalence classes partition the expressions so that the symbolic names that appear in the expressions in different equivalence classes are distinct. Thus, an expression in one equivalence class in no way constrains the values that can be assigned to the symbolic names that appear in the expressions in another equivalence class.

The equivalence classes are created by CONMAN as the constraints are recorded in the constraint structure. When an inconsistent configuration is generated, the equivalence classes of the OR-clauses that are causing the inconsistency can often be identified. Only the OR-clauses that are actually selected by CONF need be considered; and these are called the <u>active</u> OR-clauses. RECON skips over configurations that do not effect the active OR-clauses in the appropriate equivalence classes.

This section discusses the reconfiguration algorithm in detail. The first subsection describes the manner in which CONMAN maintains the equivalence classes and the role that they play in determining which configurations are to be skipped. The second subsection describes the full algorithm and then illustrates it with an example.

6.1. Equivalence Classes

RECON is often required to determine if the need to satisfy a particular expression in the constraint structure can affect the solutions that can be chosen for another expression in the constraint structure. This can happen directly, if the same symbolic name appears in both of the expressions, or indirectly, if there is a third expression in the constraint structure that can affect the solutions that can be chosen for both of the expressions. To facilitate this determination, CONMAN partitions the expressions in the constraint structure so that expressions in different equivalence classes cannot affect one another in this manner, but expressions in the same equivalence class can. Thus the symbolic names that occur in the expressions in different equivalence classes of this partition are distinct. Furthermore, this is the coarsest partition with this property. In general, if any set of equality and inequality expressions is partitioned into equivalence classes of this nature, a solution for the full set of expressions can be obtained by generating a solution for each partition individually.

To realize this partition of the expressions in the constraint structure, CONMAN first partitions the symbolic names that occur in the expressions. Then the equivalence class of an expression is determined by the equivalence class of the symbolic names that occur in that expression. Since the set of symbolic names is empty before the first branch condition is generated, the partition of the symbolic names in the constraint structure is initially empty. Whenever CONMAN records an equality or inequality expression in the constraint structure it notes the symbolic names that occur in the new expression. If none of the symbolic names in the new expression appear in expressions that are already recorded in the constraint structure, CONMAN creates a new equivalence class consisting of the new symbolic names. Otherwise, CONMAN merges the set of symbolic names in the new expression with all existing equivalence classes that intersect this set to form one equivalence class.

To demonstrate how equivalence class information can be used to identify the active clauses that are causing an inconsistency, it is easiest to consider an example. Assume that Figure 17 represents the constraint structure during analysis of an augmented condition. Ai has been used to represent the ith AND-constraint and Cij to represent the jth OR-clause of the ith OR-group. Assume that the symbolic names in the augmented condition belong to one of two equivalence classes, E1 or E2. The equivalence classes to which the AND-constraints and OR-clauses belong have been indicated in the diagram. Thus, E1 is

{A1, A3, A4, A5, C11, C21, C22, C31}

and E2 is

{A2, C12, C32, C33, C41, C42}.

Assume that RECON has been called from PROANS and that

	AND-structure	OR-structure	
	A1/E1	C11/E1 or C12/E2	
	A2/E2	C21/E1 or C22/E1	
	A3/E1	C31/E1 or C32/E2 or C33/E2	
OLDAN->	A4/E1	C41/E2 or C42/E2	<-OLDOR, NEWOR
NEWAN->	A5/E1		

Figure 17

Partition of Constraint Structure into two Equivalence Classes

[1, 1, 2, 1], its starting configuration, is inconsistent. Then the set of expressions involved in this configuration,

S = {A1, A2, A3, A4, A5, C11, C21, C32, C41},

is inconsistent. This set can be divided according to the equivalence classes to which its expressions belong to obtain the following two subsets:

S1 = {A1, A3, A4, A5, C11, C21}

and

S2 = {A2, C32, C41}.

Since S1 and S2 are in different equivalence classes, the symbolic names that are involved in the expressions in S1 and S2 are distinct. Thus, if S1 and S2 are both consistent a solution for S could be obtained by combining a solution for S1 with a solution for S2. Either S1 or S2, therefore, must be inconsistent. But S2 is consistent since the conjunction of the old AND-constraints and the OR-clauses that are selected by RECON's starting configuration is satisfiable. Thus S1 must be inconsistent, which implies that all configurations that select both C11 and C21 are inconsistent. The AND-constraints and the active OR-clauses that belong to the same equivalence class as the new AND-constraint are causing the inconsistency. In this example, therefore, RECON would increment CONF to [1, 2, 0, 0], the smallest configuration that is larger than the starting configuration and that does not select both C11 and C21. In general, whenever CONF is found to be inconsistent and a consistent subset of the expressions involved in the inconsistent configuration is known from previous results, equivalence class information can be used to skip over inconsistent configurations. The active OR-clauses that are causing the inconsistency must belong to the same equivalence classes as the expressions involved in the inconsistent configuration that do not appear in the known, consistent subset.

6.2. The Reconfiguration Algorithm

During reconfiguration CONF is alternately incremented and tested, until either a consistent configuration that selects an OR-clause from each of the first N OR-groups is found or all such configurations are determined to be inconsistent. This subsection describes this algorithm in more detail. To facilitate this description, _J_ is used to denote the number of the last OR-group from which CONF selects an OR-clause. Initially, J = N if the call to RECON is from PROANS, and J = N-1 if the call to RECON is from PROGP.

RECON maintains a set of equivalence classes, called the _marking set_, which consists of the equivalence classes of the active OR-clauses that could possibly be causing the inconsistency. Those OR-clauses that are active and that belong to equivalence classes in the marking set are said to be _marked_. In the example in Figure 17, the marking set would contain E1, marking C11 and C21 as the only OR-clauses that could possibly be causing the inconsistency. RECON initializes the marking set to consist of the equivalence classes of the new AND-constraints if the call to RECON is from PROANS, or to consist of the equivalence classes of the OR-clauses in the focal group if the call to RECON is from PROGP.

The reconfiguration algorithm consists of three steps: a back up step, a testing step and an extension step. These steps are summarized in Figure 18. The back up step is used to skip over configurations that, with the help of equivalence class information, can be shown to be inconsistent. The testing step is used to test CONF for consistency once it has been incremented. The extension step is used to extend CONF to the next OR-group after it is found to be consistent. These steps are described in more detail below.

The reconfiguration algorithm begins with an application of the back up step. The back up step increments CONF until the set of active clauses no longer contains the set of marked clauses. If all configurations that are larger than the starting configuration select the set of marked clauses or if the set of marked clauses is empty, the reconfiguration algorithm terminates unsuccessfully in this step. CONF selects OR-clauses from either the same or a smaller number of OR-groups after it has been incremented in the back up step than it did before it was incremented. Thus J, the number of OR-groups from which CONF selects an OR-clause, may be "backed up" in this step.

```
(1) Back up step:

    If possible, increment CONF to the smallest
        configuration for which the set of active clauses
        does not contain the set of marked clauses,
        J=index of last OR-group with non-zero CONF entry,
        and proceed to the testing step (2);

    Otherwise, terminate unsuccessfully;

(2) Testing step: call the inequality solver to test CONF
    for consistency;

    If CONF is consistent then
        record the solution in CFSOL;

    If CONF is consistent and J < N then
        marking set = the empty set and
        proceed to the extension step (3);

    If CONF is consistent and J = N then
        terminate successfully;

    If CONF is inconsistent and CONF(J) is not the last
        OR-clause in the Jth OR-group then

        increment CONF to select the next OR-clause from the
            Jth OR-group and

        proceed to the testing step (2);

    If CONF is inconsistent and CONF(J) is the last
        OR-clause in the Jth OR-group then

        add the equivalence classes of the clauses in the
            Jth OR-group to the marking set,

        record a zero in the Jth entry of CONF, and

        proceed to the back up step (1);

(3) Extension step:  increment CONF to select the first
    OR-clause from the (J+1)th OR-group and proceed to the
    testing step (2);
```

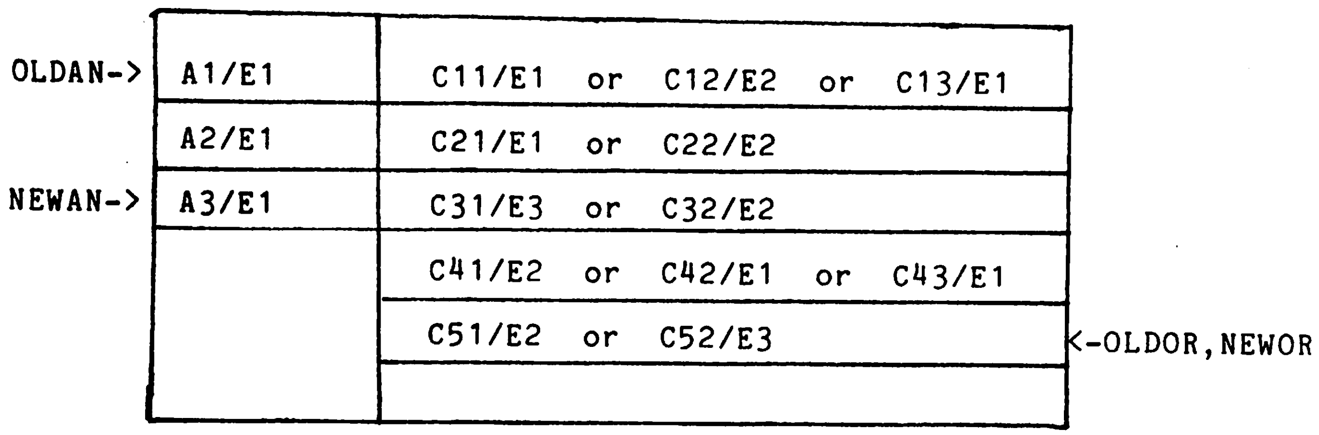
The Reconfiguration Algorithm

Figure 18

After CONF has been incremented, it may be able to be consistently extended to the first N OR-groups. Before an attempt to extend it is made, however, it is tested for consistency. Thus, RECON executes the testing step next. In the testing step the inequality solver is invoked to generate a solution for CONF. If it obtains a solution CFSOL is updated. Otherwise CONF is inconsistent.

After an application of the testing step, if CONF is consistent and J = N, the reconfiguration algorithm terminates successfully. Otherwise, the marking set is updated and CONF is reincremented. The result of the consistency test and the value of CONF determine how the marking set is updated and the value to which CONF is incremented. If CONF is inconsistent and the active OR-clause from the Jth OR-group is not the last OR-clause in that OR-group, the marking set is left unchanged and CONF is incremented to select the next OR-clause from the Jth OR-group. If CONF is inconsistent and the active OR-clause from the Jth OR-group is the last OR-clause in that OR-group, the equivalence classes of the OR-clauses in the Jth OR-group are added to the marking set, a zero is recorded in the Jth entry of CONF, and CONF is incremented by reapplying the back up step. If CONF is consistent and J < N, the marking set is emptied and CONF is incremented by applying the extension step, which increments CONF to select the first OR-clause from the (J+1)st OR-group. Once the marking set has been updated and CONF has been incremented, the testing step is reapplied. This incrementing and testing cycle continues as described above until the algorithm terminates, either successfully after the testing step or unsuccessfully in the back up step. To illustrate this process, assume that RECON is called from PROANS with N equal to 5 and a starting configuration of [1, 1, 2, 1, 1]. The constraint structure that is to be assumed is depicted in Figure 19, using the same notation as Figure 17. In this example there are three distinct equivalence classes, E1, E2 and E3. To summarize the reconfiguration, the steps that are executed and the configuration and marking set that each step starts with have been indicated in Figure 19. In addition, the marked clauses for each backup have been listed. The consistency status that is to be assumed for the configurations that are tested is indicated by a "C" for a consistent configuration and an "I" for an inconsistent configuration.

Since the new AND-constraints belong to E1, the marking set is initialized to consist of this single equivalence class. At the beginning of the first step, therefore, C11 and C21 are marked, causing CONF to be incremented to [1, 2, 0, 0, 0]. As indicated by the consistency status, this configuration is found to be consistent in the testing step, and thus, the marking set is emptied and the algorithm goes into the extension step. In three iterations of the extension and testing steps, CONF is extended until both [1, 2, 1, 1, 1] and [1, 2, 1, 1, 2] are found to be inconsistent. E2 and E3 are then added to the marking set, CONF is set back to [1, 2, 1, 1, 0], and the algorithm



Step	Starting Configuration	Initial Value of J	Marking Set	Marked Clauses	Consistency Status
(1)	(1 1 2 1 1)	5	{E1}	C11,C24	
(2)	(1 2 0 0 0)	2	{E1}		C
(3)	(1 2 0 0 0)	2	empty		
(2)	(1 2 1 0 0)	3	empty		C
(3)	(1 2 1 0 0)	3	empty		
(2)	(1 2 1 1 0)	4	empty		C
(3)	(1 2 1 1 0)	4	empty		
(2)	(1 2 1 1 1)	5	empty		I
(2)	(1 2 1 1 2)	5	empty		I
(1)	(1 2 1 1 0)	4	{E2,E3}	C22,C31,C41	
(2)	(1 2 1 2 0)	4	{E2,E3}		I
(2)	(1 2 1 3 0)	4	{E2,E3}		I
(1)	(1 2 1 0 0)	3	{E1,E2,E3}	C11,C22,C31	
(2)	(1 2 2 0 0)	3	{E1,E2,E3}		C
(3)	(1 2 2 0 0)	3	empty		
(2)	(1 2 2 1 0)	4	empty		C
(3)	(1 2 2 1 0)	4	empty		
(2)	(1 2 2 1 1)	5	empty		C

Figure 19

A Sample Reconfiguration

returns to the back up step. This time C22, C31, and C41 are marked. Thus, CONF is incremented to [1, 2, 1, 2, 0]. In the next two applications of the testing step [1, 2, 1, 2, 0] and [1, 2, 1, 3, 0] are found to be inconsistent. Thus, E1 and E2 are added to the marking set, the counter is set back to [1, 2, 1, 0, 0], and the back up step is re-executed. This time CONF is incremented to [1, 2, 2, 0, 0], which is found to be consistent in the testing step. Thus, the marking set is emptied and the algorithm begins another extension phase. CONF is extended in this phase one group at a time, to obtain [1, 2, 2, 1, 1], the smallest consistent configuration that selects an OR-clause from all five OR-groups.

7. Reduction of the Constraint Structure

Both during and after analysis of the augmented condition, reductions are made to the constraint structure that facilitate the analysis of all subsequent augmented conditions. CONMAN reduces the constraint structure in three different ways. One, it eliminates the representations of AND-constraints, OR-clauses and OR-groups that are no longer needed in order to build the path condition. Two, it solves for certain symbolic names and symplifies the representations of constraints in the constraint structure accordingly. Three, it reclaims unused space when a consolidation of the constraint structure is warranted. This section describes when and how each of these reductions are made.

When CONMAN eliminates the representation of an AND-constraint, OR-clause or OR-group from the constraint structure, the AND-constraint, OR-clause or OR-group is said to be killed. In general, elements of the constraint structure are killed in order to simplify the representation of the augmented condition. The reasons for killing a specific element, however, along with the implementation details, depend on whether the element is an AND-constraint, an OR-clause, or an OR-group.

The augmented condition can be simplified by eliminating AND-constraints that are found to be implied by other AND-constraints. Thus, new AND-constraints that are dominated by old AND-constraints are killed in the first step of PROANS. At the same time, if a new AND-constraint and an old AND-constraint are found to be replaceable by an equality, the new AND-constraint is killed and the relational operator in the representation of the old AND-constraint is changed from $\leq$ to $=$. If the augmented condition is found to be inconsistent TRYPTH restores all relational operators that were changed in the first step of PROANS back to $\leq$, before deleting the new condition from the constraint structure. If the augmented conditon is consistent, TRYPTH kills any old AND-constraints that are dominated by new AND-constraints before incorporating the new condition into the existing condition.

The manner in which an AND-constraint is killed depends on its position in the AND structure. If the constraint is the last constraint in the AND structure, it is killed by simply decrementing NEWAN. Otherwise, it is killed by replacing its coefficient vector with the zero vector and its constant term with zero. This effectively replaces the AND-constraint with a trivial equality or inequality.

If a particular OR-clause can be shown to be inconsistent with the rest of the augmented condition, the OR-clause can be eliminated from this condition. Thus, any OR-clauses in the focal group that are found to be inconsistent with the AND-constraints are killed in the first step of PROGP. Then again, in the third step of PROGP, any OR-clauses in the focal group that are found to be inconsistent with the set of AND-constraints are killed. When an OR-clause is killed, it is flagged as "dead" by negating its ORROW entry. Thus, a negative ORROW value indicates that all entries of the ORAA, ORS, ORB and ORCOL arrays that are associated with the negative ORROW entry are no longer considered to be a part of the OR structure.

If an OR-clause is implied by an AND-constraint, the entire OR-group can be eliminated from the augmented condition. Thus, if an OR-clause is found to be dominated by an AND-constraint in the first step of PROGP, the focal group is killed. When an OR-group is killed it is flagged as dead by recording the negation of the number of the dominated OR-clauses in the entry of CONF that corresponds to the OR-group. Thus, the non-negative entries of CONF define the configuration that it represents, while the negative entries flag portions of the ORAA, ORCOL, ORB and ORS arrays that are no longer considered to be a part of the OR structure.

Because OR-clauses in the focal group can be killed in the first and third steps of PROGP, the number of OR-clauses left in the focal group are counted after both of these steps. If the focal group is found to consist of a single OR-clause, the group is changed into an AND-constraint. Thus, the focal group is flagged as dead by negating the number of its one remaining OR-clause and recording the result in the focal group's CONF entry. Then, provided that the equality or inequality that defines the single OR-clause is not dominated by an AND-constraint, it is added to the bottom of the AND structure.

When an augmented condition is found to be consistent, analysis of the smallest, full, consistent configuration may indicate that some of the first few OR-groups can be changed into AND-constraints. Consider, for example, the following situation. Assume that TRYPTH finds the smallest, full, consistent configuration for an augmented condition to be [2, 3, 1, 2] and that the first and second OR-groups in this condition contain two and three OR-clauses, respectively. Then, since configurations are ordered lexicographically, any full configurations that select the first OR-clause from the first OR-group, or the first or second OR-clause from the second OR-group are inconsistent. Thus, both of these OR-groups can be changed into AND-constraints. The first can be changed into the AND-constraint that consists of the expression that defines its second OR-clause, while the second can be changed into the AND-constraint that consists of the expression that defines its third OR-clause. To change the first OR-group into an AND-constraint, a -2 is recorded in its CONF entry and, if its second OR-clause is not dominated by an AND-constraint, the OR-clause is added to the bottom of the AND structure. Similarly, a -3 is recorded in the second OR-group's CONF entry and, if its third OR-clause is not dominated by an AND-constraint, the OR-clause is added to the bottom of the AND structure.

Another opportunity for simplifying the constraint structure arises if the value of a symbolic name is completely determined by an AND-constraint. If an AND-constraint is defined by an equality that involves a single symbolic name, it can be solved to obtain the single value that every solution assigns to this symbolic name. Other constraints that involve this symbolic name can then be simplified by substituting this value in for the symbolic name. For example, if $(-2A\$1 = 3)$ and $(2A\$1 - A\$2 + 4A\$3 \leq 9)$ represent two AND-constraints in the augmented condition, -3/2 can be substituted for A$1 into the later constraint to obtain $(-A\$2 + 4A\$3 \leq 12)$. Thus, if the augmented condition is consistent CONMAN inspects the new AND-constraints for equalities that involve a single symbolic name. These equalities are used to obtain values for the symbolic names involved. The values are then used to eliminate these symbolic names from all other equalities and inequalities in the constraint structure. If this final step generates other equalities that involve a single symbolic name, the cycle is repeated.

Whenever an AND-constraint is replaced by a trivial equality or inequality or symbolic names are solved for and eliminated from expressions in the constraint structure, zero entries are introduced in the AA array. After the augmented condition has been analyzed, therefore, if a significant portion of the AA array contains zero entries, the array is consolidated by eliminating its zero entries. As each zero entry is deleted from the AA array, the corresponding entries of JCOL are deleted, subsequent non-zero entries of the AA array are shifted forward, as are the corresponding entries of JCOL, and IROW is updated appropriately. After this consolidation, the AA array records only the non-zero coefficients of the AND-constraints in the simplified augmented condition.

8. Detecting Possible Program Errors

CONMAN is able to detect possible program errors and generate test data that would cause these errors by slightly modifying the procedure that it uses to maintain the partial path condition. An error can occur at the nth node of a path, Pk, if the conjunction of the partial path condition for Pk\n and a temporary condition, generated by ATTEST at node n to represent an error, is consistent. Thus, after Pk\n has been symbolically executed and before the partial path is extended, CONMAN determines if the error can occur.

CONMAN first records the temporary condition in the new section of the constraint structure, just as if it represented a new branch condition. At this point, since the old section of the constraint structure contains the partial path condition for Pk\n, the constraint structure represents the condition under which Pk\n would be executed and the error would occur.

CONMAN then invokes TRYPTH to analyze the condition recorded in the constraint structure, thereby determining if the temporary condition is consistent with the existing condition. TRYPTH analyzes the condition in the constraint structure exactly as if it represented a partial path condition. Thus, TRYPTH starts with the configuration and solution that were recorded in EXCON and EXSOL in the previous call to TRYPTH, and if the temporary condition is consistent with the existing condition, it records the smallest, full, consistent configuration and a corresponding solution in CONF and CFSOL, as described in Section 4.

CONMAN then updates the constraint structures so that symbolic execution of the program path may continue. It is in this step that the procedure for maintaining the partial path condition differs from the procedure for detecting program errors. Whereas TRYPTH records the values of CONF and CFSOL in EXCON and EXSOL if a partial path condition is found to be consistent, it does not alter EXCON and EXSOL after determining if a temporary condition is consistent with the existing condition. Recall that EXCON and EXSOL are used in the next call to TRYPTH to initialize CONF and CFSOL. Thus, the next time it is called, TRYPTH starts with the smallest, consistent configuration for Pk\n and a corresponding solution. After analyzing the condition in the constraint structure, if the new condition is a temporary condition TRYPTH restores the constraint structure to once again represent the existing condition. Thus, the relational operators of any constraints that were changed from $\leq$ to = in the first step of PROANS are changed back to $\leq$, and NEWAN and NEWOR are repositioned to point to the bottom of the old sections of the constraint structure. In effect, the repositioning of NEWAN and NEWOR deletes the temporary condition, along with any constraints that were added to the bottom of the AND structure in the first step of PROGP. Since only new AND-constraints and new OR-groups can be killed in the first steps of PROANS and PROGP, no further modifications need to be made to the constraint structure at this time. Recall that, after analyzing a partial path condition that is found to be consistent, TRYPTH kills old AND-constraints that are dominated by new ones. As it does if the partial path is found to be inconsistent, TRYPTH omits this modification when called with a temporary condition instead of a branch condition.

Finally, if the temporary condition is found to be consistent with the existing condition, TRYPTH returns the solution recorded in CFSOL. This provides a data set that would cause the program error. If the temporary condition is inconsistent with the existing condition, TRYPTH returns a flag indicating that the error cannot occur.

Appendix A: The BNB structure

The data structure that defines the problem to be solved by the inequality solver is called the BNB structure. When the inequality solver is to be used to test a configuration for consistency, the BNB structure must contain the AND-constraints and the OR-clauses that are selected by the configuration. Since the AND structure is a part of the BNB structure, the BNB structure always contains the AND-constraints. The appropriate OR-clauses, however, have to be entered into the BNB structure before the inequality solver is called. OR-clauses are recorded in the section of the BNB structure that is referred to as the effective OR-clause structure. The OR-clauses selected by a configuration are recorded in the effective OR-clause structure in the order in which the OR-groups to which they belong are recorded in the OR structure.

The AA, JCOL, IROW, S and B arrays are used by both the AND structure and the effective OR-clause structure. The AND-constraints are recorded as the first NEWAN equalities and inequalities in these arrays. The equalities and inequalities in the effective OR-clause structure are recorded sequentially in these same arrays, immediately following the AND-constraints. The global variable, M, points to the bottom of the BNB structure. Thus, the effective OR-clause structure consists of the (NEWAN + 1)st through the Mth equalities and inequalities that are recorded in the AA, JCOL, IROW, S and B arrays. This arrangement of the BNB structure is depicted in Figure 20.

BNB Structure

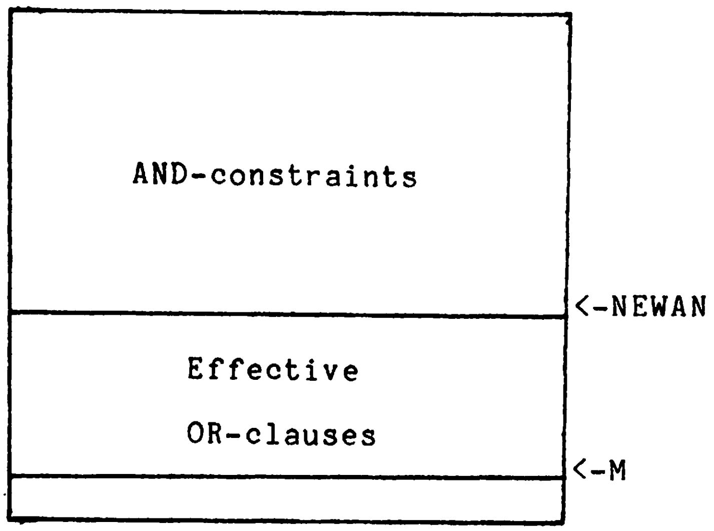


Figure 20

Division of the BNB Structure into its two Components:
AND-structure and Effective OR-clause Structure