A Formal Basis For Removing *Goto* Statements

SI PAN AND R. GEOFF DROMEY

Software Quality Institute, Griffith University, Brisbane, QLD. 4111, Australia
Email: G.Dromey@cit.gu.edu.au

***Goto* statements detract from the quality of imperative programs. They tend to make control-structures difficult to understand and, at the same time, introduce the risk of non-termination and other correctness problems. A new, formal, generally applicable procedure for removing all *goto* statements from program structures is presented. This method is based on formal semantics and congruent equivalence transformations. Not only does the method logically simplify program structures; it also detects a range of defects including a class of non-termination problems, unreachable code and redundancy problems. The method can also be used to eliminate recursion.**

Received March 29, 1994; revised March 20, 1996

1. INTRODUCTION

Imperative program components containing ***goto*** statements are usually regarded as very difficult structures to analyse, modify, restructure and prove correct [1]. It has long been recognized that the use of ***goto*** statements significantly detracts from the structural integrity, simplicity, reliability and the ultimate quality of programs.

Although the use of ***goto*** statements became unfashionable more than two decades ago, they are still found in some programs that must be re-engineered and maintained. An examination of the literature in this area over the past two decades reveals that a number of studies [2–11] have been done on methods for eliminating ***goto*** statements and recursion [12, 13] from programs. Assessing these transformations, methods and results we conclude that:

- those that involve transformations heavily rely on pattern matching and therefore tend to lack generality;
- they sometimes result in transformations that change properties of the original programs;
- they often introduce logical and textual inefficiencies.

There appears to be no powerful, widely applicable, formal means for removing ***goto*** statements from programs. What we will describe here is a new, formally based, systematic method for removing ***goto*** statements from sequences and loop structures. In our treatment we assume a ***goto*** statement always has associated with it a ***label*** which determines the next statement to be executed when the ***goto*** is reached.

The proposed method has many similarities to the way we use substitution to solve a set of algebraic equations. A closed structure involving one or more ***goto*** statements and a set of ***label*** statements is mapped into a set of *statement equations*. The 'solution' of this set of equations is derived using semantics-preserving substitutions. Several key semantics-preserving transformations are also employed (they may be likened to algebraic simplification). Together these substitutions and transformations provide a formal and systematic means for removing ***goto*** statements and, at the same time, optimally restructuring the original delinquent program fragment.

The method offers a rigorous means for re-engineering existing, poorly structured, legacy programs into improved structures that satisfy their original specification. Strongest postcondition calculations may be used to prove that all the transformations employed yield equivalent program components [14–16]. An important feature of the method, apart from its restructuring capability, is that it can detect a significant class of termination problems. The method is easy to apply manually and it is amenable to automation.

2. TRANSFORMING EXITING *GOTO*S INTO INTERNAL *GOTO*S

Two structurally different categories of ***goto*** statements may be encountered. We will first consider ***goto*** statements in loops and subsequently generalize the treatment needed to handle other structural contexts. For loops, the first category is distinguished by the fact that the associated label statement is still within the same loop body. This is called an *internal **goto** statement.* The second category of ***goto*** statement is one that is used to transfer control out of a loop—in this case the label is external to the loop. It is called an *exiting **goto** statement.* The concept of internal/exiting ***goto***s is not limited to loop bodies. It can be extended to model closed blocks. If we can remove both these categories of ***goto*** we can eliminate all ***goto***s from any program structure.

In this section we will deal with removing exiting ***goto*** statements. We assume any exiting ***goto*** statement in a given loop body is *guarded*. If it is not, it must occur in a sequential loop body of the form ***do*** $G \rightarrow S;$ ***goto*** $L;\ S'$ ***od***, where the label L does not appear in S or S'. The control-flow after execution of S will terminate the iteration and S' will never be executed. In this situation, the loop should be replaced by a branch statement ***if*** $G \rightarrow S;$ ***goto*** L ***fi*** and the redundant statement sequence S' should be removed. In the following we always treat any exiting ***goto*** as a guarded exiting ***goto***.

Let us consider a loop with a guarded exiting ***goto***

statement, denoted by ***do*** $G \rightarrow S$; ***if*** $C \rightarrow S_1$; ***goto*** L; $S_2 [] \neg C \rightarrow S_3$ ***fi***; S' ***od***. The function of this ***goto*** is to transfer control out of the loop. It can be replaced by a ***break*** statement which terminates the loop (note a ***break***, which is semantically equivalent to a specialized forward-only ***goto***, is much easier to remove—see below). After termination the control-flow needs to be transferred to the statement labelled L. This suggests implementation by a statement ***goto*** L placed immediately following the loop. The problem with this is that if we use the desired transformation ***do*** $G \rightarrow S$; ***if*** $C \rightarrow S_1$; ***break*** $[] \neg C \rightarrow S_3$; S' ***fi od***; ***goto*** L, the normal exit (at the loop guard) will, on termination, also transfer control to the statement labelled by L. A guard is therefore needed to distinguish the different exits. The simplest way to achieve this is to introduce a fresh *boolean* variable *jump* and initialize it to false. Then, if the ***goto*** L is to be executed the *jump* must be set to true prior to executing the ***break*** that has been introduced into the loop. We then have:

Rule for removal of a guarded exiting ***goto*** (REG):

do $G \rightarrow S$; ***if*** $C \rightarrow S_1$; ***goto*** L $[] \neg C \rightarrow S_3$; ***fi***; S' ***od***
$\models$ *jump* := false;
do $G \rightarrow S$; ***if*** $C \rightarrow S_1$; *jump* := true; ***break***
$[] \neg C \rightarrow S_3$; S' ***fi*** *od*;
if *jump* $\rightarrow$ ***goto*** L ***fi***

where *jump* is a fresh variable

All exiting ***goto***s can be removed from a loop by use of ***break***s and fresh variables. For example, consider a loop with a nested subloop which contains a guarded exiting ***goto*** statement:

do $G \rightarrow S$; ***do*** $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$;
goto L; S_2 $[] \neg C \rightarrow S_3$ ***fi***; S'' ***od***; S''' ***od***

After application of REG twice, we can convert the exiting ***goto*** statement into an internal ***goto*** statement, that is:

Therefore, to remove all exiting ***goto***s from a loop body we first employ the REG transformation as many times as necessary. We start the process by removing ***goto***s from internal nested subloops. The newly produced guarded ***goto***s then become guarded ***goto***s for the external loops. Repeating this process, all exiting ***goto***s including the newly introduced ***goto***s will finally be removed from the transformed loop structure. All occurrences of exiting ***goto***s will then be replaced by ***break***s with all exiting ***goto***s assuming the status of internal ***goto***s. Subsequently the process of removing the flags that have been introduced and the ***break***s can be accomplished by loop rationalization which is discussed in the companion paper [17]. Finally, the newly produced internal ***goto***s can be removed by the method we will describe in the rest of this paper.

3. THEORETICAL BASIS FOR REMOVING *GOTO* STATEMENTS

In this section we introduce a formal method that is suitable for eliminating ***goto***s from any program structure. Since the elimination of exiting ***goto***s has been dealt with, the remaining task is to handle the elimination of ***goto***s from a sequence. This corresponds to elimination of internal ***goto***s from a given closed block (including a loop body).

Initially we will assume all ***goto***s to be processed are not exiting to a guarded structure. A statement ***goto*** L is called *a* **goto** *statement exiting to a guarded structure* if the label L occurs in a guarded statement, such as ***if*** $C \rightarrow \ldots$ L: $\ldots [] \neg C \rightarrow \ldots$ ***fi*** or ***do*** $G \rightarrow \ldots$ L: $\ldots$ ***od***. The case of ***goto***s exiting to a guarded structure will be dealt with later in section 3.4. All labels corresponding to internal ***goto***s here will therefore occur in a sequential structure.

3.1. Statement variables and statement variable equations

To reason systematically and formally about structures that contain internal ***goto***s it is necessary to introduce some formal structures. We borrow algebraic concepts of equation and solution to describe this process. *Statement variable equations* play the most important role. We will show how to construct these equations from program code,

do $G \rightarrow S$; ***do*** $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$; ***goto*** L; $S_2 [] \neg C \rightarrow S_3$ ***fi***; S'' ***od***; S''' ***od***
$\models$ ***do*** $G \rightarrow S$;
jump$'$:= false;
do $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$; *jump*$'$:= **true**; ***break*** $[] \neg C \rightarrow S_3$ ***fi***; S'' ***od***;
if *jump*$'$ $\rightarrow$ ***goto*** L ***fi***;
S'''
od (applying REG for the subloop)
$\models$ *jump* := false;
do $G \rightarrow S$;
jump$'$:= false;
do $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$; *jump*$'$:= **true**; ***break*** $[] \neg C \rightarrow S_3$ ***fi***; S'' ***od***;
if *jump*$'$ $\rightarrow$ *jump* := **true**; ***break*** $[] \neg$ *jump*$'$ $\rightarrow S'''$ ***fi***
od;
if *jump* $\rightarrow$ ***goto*** L ***fi*** (applying REG for the outer loop)

and how to reason with them to generate fruitful restructured results.

Consider a sequence containing n labels, denoted by S_0; ***label***$_1$: S_1; ***label***$_2$: S_2; ...; ***label***$_n$: S_n. We may introduce n *statement variables* [14] X_i where $i \in [1, n]$. X_i denotes a statement sequence commencing from the statement S_i and extending to the end of the sequence. This means $X_i = S_i$; ***label***$_{i+1}$: S_{i+1}; ...; ***label***$_n$: S_n for $i \in [1, n-1]$ and $X_n = S_n$. Schematically shown in Figure 1.

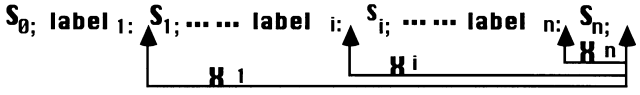

FIGURE 1.

From these relations it follows that $X_i = S_i$; X_{i+1} for $i \in [1, n-1]$ and $X_n = S_n$. Since the block that is executed after execution of ***goto label***$_j$ is X_j, where $j \in [1, n]$, we can build a *statement variable equation* set [18] consisting of n equations of the form:

$$X_i = (S_i;\ \textbf{\textit{label}}_{i+1}:\ S_{i+1})\ [|\textbf{\textit{label}}_1/X_1, \textbf{\textit{label}}_2/X_2, \ldots, \textbf{\textit{label}}_n/X_n|] \quad \text{for any } i \in [1, n-1]$$

$$\text{and } X_n = S_n[|\textbf{\textit{label}}_1/X_1, \textbf{\textit{label}}_2/X_2, \ldots, \textbf{\textit{label}}_n/X_n|]$$

where the notation $S[|\textbf{\textit{label}}_1/X_1, \textbf{\textit{label}}_2/X_2, \ldots, \textbf{\textit{label}}_n/X_n|]$ means substitution of the earliest occurring unguarded sequence that starts with ***goto label***$_i$; ... or ***label***$_i$: S_i; ... by X_i for each branch of S. Note S_i may contain **gotos** but no labels. To understand this substitution, let us consider the three possible situations:

- where S_i does not contain any ***goto***, then $X_i = S_i$; X_{i+1} or $X_n = S_n$;
- where S_i contains an unguarded ***goto label***$_j$, i.e., $S_i = SS_i$; ***goto label***$_j$; SS_{i+1}, then

  $X_i = SS_i; X_j$ where SS_i does not involve any ***goto***s;
- otherwise S_i may always be represented, in general, by

  SS_0; ***if*** $C_1 \rightarrow SS_1$ $[]\, C_2 \rightarrow SS_2\, [] \ldots []\, C_m \rightarrow SS_m$ ***fi***; SS_{m+1}

  where SS_0 does not involve any ***goto***s, and $\forall i \forall j\ ((i, j \in [1, m] \wedge j \neq i) \Rightarrow (C_i \Rightarrow \neg C_j))$ and $C_1 \vee C_2 \vee \ldots \vee C_m \equiv$ true, then the equation is recursively defined as:

$$\begin{aligned} X_i = SS_0;\ \textbf{\textit{if}}\ & C_1 \rightarrow (SS_1; SS_{m+1}; \textbf{\textit{label}}_{i+1}:\ S_{i+1}) \\ & [|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|] \\ & []\, C_2 \rightarrow (SS_2; SS_{m+1}; \textbf{\textit{label}}_{i+1}:\ S_{i+1}) \\ & [|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|] \\ & [] \ldots \\ & []\, C_m \rightarrow (SS_m; SS_{m+1}; \textbf{\textit{label}}_{i+1}:\ S_{i+1}) \\ & [|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|] \\ & \textbf{\textit{fi}}. \end{aligned}$$

These definitions precisely capture the semantics of statement variables. The equation describes recursive relations among the n statement variables. It also describes the control-flow behaviour during execution of this sequence. For example, the equation $X_i = SS_i$; ***if*** $C \rightarrow SS_j$; X_j ***fi*** indicates that after execution of SS_i the control-flow checks the condition C. If it holds, then SS_j executes followed by the statements corresponding to X_j. If C does not hold, termination will take place. An equation of the form $X = S$; ***if*** $C \rightarrow S'; X$ ***fi*** indicates a loop structure for X. According to the semantics it may therefore be replaced by $X = S$; ***do*** $C \rightarrow S'$; S ***od***.

When a statement variable equation is of the form:

$$\begin{aligned} X_i = SS_0;\ \textbf{\textit{if}}\ & C_1 \rightarrow SS_1;\ X_{i_1}\, [] \ldots []\, C_{t-1} \rightarrow SS_{t-1}; X_{i_{t-1}}; \\ & []\, C_t \rightarrow SS_t\, [] \ldots []\, C_m \rightarrow SS_m\ \textbf{\textit{fi}} \end{aligned}$$

where SS_j does not involve any statement variables or ***goto***s, $j \in [1, m]$ and $m \geqslant 0$,

we call it a *standard equation*. It encompasses the simplified forms $X_i = SS_0$; SS_p, $X_i = SS_0$; SS_q; X_q and $X_i = SS_0$; ***if*** $C_q \rightarrow SS_q$; X_q ***fi***, where $p \in [t, m]$ and $q \in [1, t-1]$.

3.2. Sequentializing nested selections

Some transformations are needed to convert statement variable equations directly from raw code into standard equations that are easier to apply substitution to and to reason about. Our approach is similar to the strategy used to solve individual and sets of algebraic equations. For example, we need to transform the equation $x = a^*(b - c^*(x + d))$ into the form $x = a^*b - a^*c^*x - a^*c^*d$ as a preliminary step to solving it for x, i.e. $x = (a^*b - a^*c^*d)/(1 + a^*c)$.

The notation $S[|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|]$ has a number of obvious properties. First, each terminal branch in $S[|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|]$ either contains no statement variable or it ends at a single statement variable. Secondly, in each branch of $S[|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|]$, if a statement variable is guarded, it may be guarded by a number of branch statements. The following example illustrates this:

if $C \rightarrow S;$
  if $C' \rightarrow S_1;\ X_1$
  $[]\neg C' \rightarrow S_2$
  fi
$[]\neg C \rightarrow S_3;\ X_3$
fi

To handle structures like this we must first promote all statement variables in the nested branches to their highest possible level in the statement variable equation and then form a standard equation. This requirement can be described

as a transformation from a statement of the form:

$$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C_2 \rightarrow S_2\\ \qquad\qquad [\,]\neg C_2 \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ [\,]\neg C_1 \rightarrow S_4;\\ \qquad\qquad \textbf{\textit{if}}\ C_3 \rightarrow S_5\\ \qquad\qquad [\,]\neg C_3 \rightarrow S_6\\ \qquad\qquad \textbf{\textit{fi}}\\ \textbf{\textit{fi}}\end{array}$$

into an equivalent statement that has one or the other of the following forms:

$$\begin{array}{l}\textbf{\textit{if}}\ C_1' \rightarrow S_1;\ S_2\\ [\,]\,C_2' \rightarrow S_1;\ S_3\\ [\,]\,C_3' \rightarrow S_4;\ S_5\\ [\,]\,C_4' \rightarrow S_4;\ S_6\\ \textbf{\textit{fi}}\end{array}\quad \text{or} \quad \begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1\\ [\,]\,\neg C_1 \rightarrow S_4\\ \textbf{\textit{fi}};\\ \textbf{\textit{if}}\ C_1' \rightarrow S_2\\ [\,]\,C_2' \rightarrow S_3\\ [\,]\,C_3' \rightarrow S_5\\ [\,]\,C_4' \rightarrow S_6\\ \textbf{\textit{fi}}\end{array}$$

Several manipulations allow us to complete these conversions. One of these equivalence transformation rules [16] is:

- $$\begin{array}{l}\textbf{\textit{if}}\ C \rightarrow S_1;\ S\\ [\,]\neg C \rightarrow S_2;\ S\\ \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\textbf{\textit{if}}\ C \rightarrow S_1\\ [\,]\neg C \rightarrow S_2\\ \textbf{\textit{fi}};\\ S\end{array}$$

Another transformation is:

- $$\begin{array}{l}\{P\}\ \textbf{\textit{if}}\ C \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C' \rightarrow S_2\\ \qquad\qquad [\,]\neg C' \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ \qquad [\,]\neg C \rightarrow S_4;\ S_2\\ \qquad \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\{P\}\ \textbf{\textit{if}}\ C \rightarrow S_1\\ \qquad [\,]\neg C \rightarrow S_4\\ \qquad \textbf{\textit{fi}};\\ \qquad \textbf{\textit{if}}\ C' \rightarrow S_2\\ \qquad [\,]\neg C' \rightarrow S_3\\ \qquad \textbf{\textit{fi}}\end{array}$$

where the *strongest postcondition* calculation yields $sp(P \wedge \neg C, S_4) \Rightarrow C'$, for the precondition P.

The *weakest precondition* calculations [16] yield:

- $$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C_2 \rightarrow S_2\\ \qquad\qquad [\,]\neg C_2 \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ \qquad [\,]\neg C_1 \rightarrow S_4;\\ \qquad\qquad\qquad \textbf{\textit{if}}\ C_3 \rightarrow S_5\\ \qquad\qquad\qquad [\,]\neg C_3 \rightarrow S_6\\ \qquad\qquad\qquad \textbf{\textit{fi}}\\ \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\textbf{\textit{if}}\ C_1 \wedge wp(S_1, C_2) \rightarrow S_1; S_2\\ [\,]\,C_1 \wedge \neg wp(S_1, C_2) \rightarrow S_1; S_3\\ [\,]\neg C_1 \wedge wp(S_4, C_3) \rightarrow S_4; S_5\\ [\,]\neg C_1 \wedge \neg wp(S_4, C_3) \rightarrow S_4; S_6\\ \textbf{\textit{fi}}\end{array}$$

As part of these transformations it is sometimes necessary to move an assignment or some other statement inside a guarded statement. In our companion paper on loop reengineering [17] we describe how to do this using strongest postcondition calculations. The following example illustrates the required transformation:

$$\begin{array}{l}x := x + y^*z;\ \textbf{\textit{if}}\ x - y > 0 \rightarrow y := \ldots\\ \qquad\qquad [\,]x - y \leqslant 0 \rightarrow z := \ldots\ \textbf{\textit{fi}}\\ |= \textbf{\textit{if}}\ x + y^*z - y > 0 \rightarrow x := x + y^*z;\ y := \ldots\\ \qquad [\,]x + y^*z - y \leqslant 0 \rightarrow x := x + y^*z;\ z := \ldots\ \textbf{\textit{fi}}\end{array}$$

Referring back to the original form introduced above, when $S_1 = \textbf{\textit{skip}}$, $wp(S_1, C_2) \equiv C_2$ and $wp(S_1, \neg C_2) \equiv \neg C_2$ the transformation rule above becomes:

$$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow \textbf{\textit{if}}\ C_2 \rightarrow S_2\ [\,]\neg C_2 \rightarrow S_3\ \textbf{\textit{fi}}\ [\,]\neg C_1 \rightarrow S_4; \ldots \textbf{\textit{fi}}\\ |= \textbf{\textit{if}}\ C_1 \wedge C_2 \rightarrow S_2\ [\,]\,C_1 \wedge \neg C_2 \rightarrow S_3\ [\,]\neg C_1 \rightarrow S_4; \ldots \textbf{\textit{fi}}\end{array}$$

The weakest preconditions $wp(S_1, C_2)$ or $wp(S_4, C_3)$ may be difficult to calculate (when S_1 or S_4 involve loop structures) or even undefined [when S_1 or S_4 involve a ***read*(x)** and the branch guards C_2 or C_3 also involve the variable **x**] for arbitrary input data. The following rule can, however, always realize the required transformation:

Sequentializing Nested Selection (SNS):

$$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C_2 \rightarrow S_2\\ \qquad\qquad [\,]\neg C_2 \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ [\,]\neg C_1 \rightarrow S_4;\\ \qquad\qquad \textbf{\textit{if}}\ C_3 \rightarrow S_5\\ \qquad\qquad [\,]\neg C_3 \rightarrow S_6\\ \qquad\qquad \textbf{\textit{fi}}\\ \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\ \textit{flag} := \text{true}\\ \qquad [\,]\neg C_1 \rightarrow S_4;\ \textit{flag} := \text{false}\\ \qquad \textbf{\textit{fi}};\\ \qquad \textbf{\textit{if}}\ C_2 \wedge \textit{flag} \rightarrow S_2\\ \qquad [\,]\neg C_2 \wedge \textit{flag} \rightarrow S_3\\ \qquad [\,]\,C_3 \wedge \neg\, \textit{flag} \rightarrow S_5\\ \qquad [\,]\neg C_3 \wedge \neg\, \textit{flag} \rightarrow S_6\\ \qquad \textbf{\textit{fi}}\end{array}$$

where *flag* is a fresh variable

As for REG, we may remove *flag*s using the technique described in the companion paper [17]. The process of converting any non-standard equation into a standard equation may involve a number of applications of sequentializing nested selections (SNS).

We also require that no statement variable X occurs more than once on the right-side of any standard equation. A disjunctive rule can always be used to eliminate duplicate occurrences. The example shown below illustrates the

transformation:

$$\textbf{\textit{if}}\ C_1 \to S_1; X[]\, C_2 \to S_2; X[]\, C_3 \to S_3[] \ldots []C_p \to S_p\ \textbf{\textit{fi}}$$
$$\models \textbf{\textit{if}}\ C_1 \vee C_2 \to \textbf{\textit{if}}\ C_1 \to S_1[]C_2 \to S_2\ \textbf{\textit{fi}}; X\ []C_3 \to S_3[] \ldots []C_p \to S_p\ \textbf{\textit{fi}}$$

3.3. Non-recursive solutions for standard equations

In the following sections, we will use a graph theoretic form to describe a systematic process for finding solutions for sets of standard equations. This form not only indicates detailed steps for resolution but it also suggests how the whole approach may be implemented.

3.3.1. Variable dependency graph

Consider a set of standard equations:

$$X_i = SS_0; \textbf{\textit{if}}\ C_1 \to SS_1; X_{i_1}[] \ldots []\, C_{t-1} \to SS_{t-1}; X_{i_{t-1}} []\, C_t \to SS_t[] \ldots []\, C_m \to SS_m\ \textbf{\textit{fi}}$$

where $i \in [1,n]$ and $i_j \in [1,m]$ for any $j \in [1,t-1], t \geqslant 1$.

Each equation corresponds to a node in a directed graph. Nodes are connected according to their variable dependency relations, i.e. the directed graph is formally defined as $\langle \{X_1, X_2, \ldots, X_n\}, E\rangle$, where E contains all edges $\langle X_i, X_s\rangle$ such that SS_0; ***if*** $C_1 \to SS_1$; $X_{i_1}[] \ldots []\ C_{t-1} \to SS_{t-1}$; $X_{i_{t-1}}[]\, C_t \to SS_t[] \ldots []\, C_m \to SS_m$ ***fi*** involves X_s, i.e. $X_s \in \{X_{i_1}, X_{i_2}, \ldots X_{i_{t-1}}\}$. The node X_i has $t-1$ children $\{X_{i_1}, X_{i_2}, \ldots X_{i_{t-1}}\}$.

Obviously any leaf node X_{leaf} that is not reflexive (i.e. it does not have a graph theoretic loop edge attached) indicates that the corresponding equation is $X_{leaf} = S_{leaf}$ where S_{leaf} does not involve any statement variable X_i, $i \in [1,n]$. We call such a S_{leaf} the *Solution* of X_{leaf} and the leaf X_{leaf} is said to be coloured. Generally any node is said to be coloured if we have obtained an equivalent ***goto***-free solution for its corresponding statement variables. All non-cycle leaves in the directed graph may be treated as coloured.

When all the children $\{X_{i_1}, X_{i_2}, \ldots X_{i_{t-1}}\}$ of a node X_i are coloured, it too can be coloured. The colouring of X_i maps in the equation domain to a substitution that will yield a solution for X_i. The substitution rule is described below.

Rule of Substitution Solution (RSS):

Given any (standard) statement equation

$$X_i = SS_0; \textbf{\textit{if}}\ C_1 \to SS_1; X_{i_1}[] \ldots []C_{t-1} \to SS_{t-1}; X_{i_{t-1}} []C_t \to SS_t[] \ldots []C_m \to SS_m\ \textbf{\textit{fi}}$$

and all the solutions Y_{ij} to X_i's children such that $X_{i_j} = Y_{i_j}$, $j \in [1,t-1]$, then the segment

$$SS_0; \textbf{\textit{if}}\ C_1 \to SS_1; Y_{i_1}[] \ldots []\, C_{t-1} \to SS_{t-1}; Y_{i_{t-1}} []\, C_t \to SS_t[] \ldots []\, C_m \to SS_m\ \textbf{\textit{fi}}$$

is the *Solution* for X_i.

The rule of substitution solution (RSS) suggests that we can easily obtain solutions for all statement variables. A sequence of substitutions map their statement equations to a tree(s) (without any cycle-nodes). This tree-colouring process corresponds to an extended *Post-Order* traversal, first colouring the leaves, then proceeding progressively up the tree to the root. Obviously, these solutions are ***goto***-free code segments that are semantically equivalent to their corresponding statement variables. Each colouring step defines a semantically equivalent ***goto***-free solution for a node.

The RSS rule also provides a post-order colouring strategy for any directed graph that is based on statement equations. It follows that if we can also colour cycles, we can then colour any directed graph, and obtain equivalent ***goto***-free solutions for the statement equations.

3.3.2. Self-cycle removal (LE)

Any cycle is either a self-cycle involving a single node or a complex cycle that involves more than one node. Given any self-cycle node X, its corresponding statement equation is, in general, of the form $X = S$; ***if*** $C \to S'$; $X\,[]\neg C \to Y$ ***fi***. Its semantics indicates that after execution of S the control-flow iteratively executes S'; S until the condition C fails then Y is executed. We therefore have the following colouring rule for LE:

Loop Extraction (LE):
Given any (standard) statement equation $X = S$; ***if*** $C \to S'; X\,[]\neg C \to Y$ ***fi***, the statement S; ***do*** $C \to S'$; S ***od***; Y is the *Solution* for X.

The equation $X = S; S'; X$, (corresponding to $C = \text{true}$ in LE) has a non-terminating solution S; ***do*** true $\to S'$; S ***od*** (or ***do*** true $\to$ S; S' ***od***). This form can be used to detect non-termination defects.

Using SNS, RSS and LE we can always complete the task of finding a solution for any directed graph that contains only self-cycles. From a graph theoretic standpoint, the rule LE absorbs any self-cycle node into a super-node. Therefore application of LE yields a new directed graph that is a tree(s) without any cycles. The latter can be coloured using RSS.

3.3.3. Cycle colouring

Before considering complex cycle colouring we need to review the rules RSS and LE. Given any standard equation $X = S$; ***if*** $C \to S'$; $X_a\,[]\neg C \to S''$; X_b ***fi***, RSS indicates that the substitution for X_a and X_b by their solutions Y_a and Y_b yields an equivalent equation $X = S$; ***if*** $C \to S'$; $Y_a\,[]\neg C \to S''$; Y_b ***fi*** (at this time S; ***if*** $C \to S'$; $Y_a\,[]\neg C \to S''$; Y_b ***fi*** contains no ***goto***s). Furthermore, if we substitute for X_a and X_b using the equations $X_a = S_a$; ***if*** $C_a \to S_a'$; $X_{a1}\,[]\neg C_a \to S_a''$; X_{a2} ***fi*** and $X_b = S_b$; ***if*** $C_b \to S_b'$; $X_{b1}\,[]\neg C_b \to S_b''$; X_{b2} ***fi***, the resulting substitutions should still maintain the original

equation, i.e.,

$$
\begin{aligned}
X = S;\ \textbf{\textit{if}}\ C \rightarrow S';\ S_a;\\
&\textbf{\textit{if}}\ C_a \rightarrow S'_a;\ X_{a1}\\
&[\,]\neg C_a \rightarrow S''_a;\ X_{a2}\\
&\textbf{\textit{fi}}\\
[\,]\neg C \rightarrow S'';\ S_b;\\
&\textbf{\textit{if}}\ C_b \rightarrow S'_b;\ X_{b1}\\
&[\,]\neg C_b \rightarrow S''_b;\ X_{b2}\\
&\textbf{\textit{fi}}.\\
\textbf{\textit{fi}}
\end{aligned}
$$

After sequentializing the nested selection, i.e. standardizing the substituted equations, we obtain a new standard equation for X which involves the statement variables X_{a1}, X_{a2}, X_{b1} and X_{b2}.

The role of RSS is to convert the statement-variable-dependent relation of a standard equation. That is, after applying RSS and SNS, the equation for X should only depend on the statement variables which are the child nodes (i.e., X_{a1}, X_{a2}, X_{b1} and X_{b2}) of X's direct child nodes (X_a and X_b). Similarly, the LE rule also applies for equations in the form $X = S;\ \textbf{\textit{if}}\ C \rightarrow S';\ X\,[\,]\neg C \rightarrow X'\ \textbf{\textit{fi}}$, where X' is a statement variable. The solution is $X = S;\ \textbf{\textit{do}}\ C \rightarrow S';\ S\ \textbf{\textit{od}}\ ;\ X'$. In summary, RSS together with SNS and LE allow us to transform by substitution a set of standard equations in a similar manner to the way we solve a set of algebraic equations.

Now let us consider a complex cycle with more than one node. We select a node X_1 as a *virtually coloured* node. This means that we treat the statement variable X_1 as a 'solution' during the following *virtual colouring* process. This step corresponds to converting the cycle into a tree structure, where all direct parents of the original node X_1 become the direct parents of the leaf X_1 (corresponding to the virtually coloured node) and the original node X_1 also becomes a root of the tree structure (see Figure 2).

When this resulting tree structure contains no complex cycles, all nodes can be coloured by LE and RSS. However, their solutions still contain the statement variable X_1. For this reason we call their solutions *virtual solutions* and this colouring process a *virtual colouring*. The third step is to actually colour the node X_1. Because the root's equation, after virtual colouring, contains X_1 only, we can use LE to obtain an actual solution for X_1. After substituting the actual solution for X_1 for all virtual solutions, we obtain the actual solutions for all other nodes on the cycle. When the resulting tree structure contains other complex sub-cycles, the first step (selecting a virtually coloured node) and the second step (virtually colouring) may need to be applied recursively. The third step (actually colouring) then needs to proceed progressively up the tree to the root X_1. This process is similar to the process of using the algebraic resolution method to find a solution for a set of algebraic equations $X_i = f_i(X_1, X_2, \ldots, X_m), i \in [1, m]$.

To illustrate the process of complex cycle colouring let us consider a three-node complete directed graph as an example (see Figure 3). We use a set of arbitrary equations $X_i = f_i(X_1, X_2, X_3), i \in [1, 3]$, as an example to show the process of colouring a complex cycle (see Table 1).

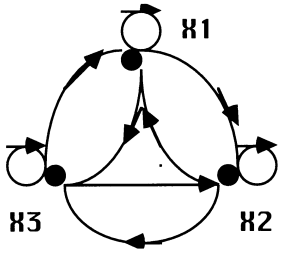

FIGURE 3.

In general, cycle-colouring (for more than one node) employs the following process:

Step 1: take a node as a virtually coloured node;
Step 2: apply the tree-colouring process to colour other nodes in the cycle until all nodes are (virtually) coloured. To achieve this, Step1, Step2 may need to be applied recursively;
Step 3: (actually) colour the virtually coloured nodes then actually colour all other nodes in the cycle.

Intuitively the process of complex cycle-colouring involves transforming and converting various forms of statement equation into standard ones using SNS, RSS and LE. We then apply LE (to find a real solution) as the last substitution (actually colouring all nodes). A number of strategies have been developed to improve the colouring process. Full details are provided in a separate report [18].

3.4. Removing *GOTO* statements exiting to guarded structures

We can now deal with ***goto*** statements exiting to guarded structures. For these cases, as long as we can correctly build statement variable equations, the method we have

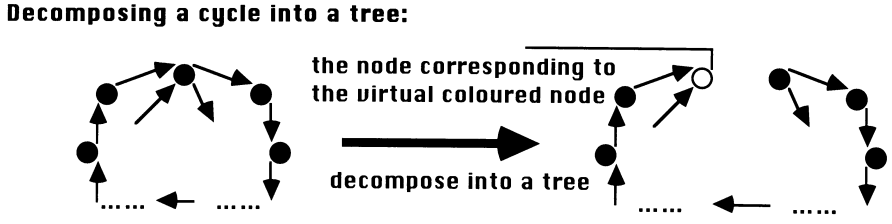

FIGURE 2.

TABLE 1

No.	*Input form*	*Output form*	*Comment*
1	$X_1 = f_1(X_1, X_2, X_3)$ $X_2 = f_2(X_1, X_2, X_3)$ $X_3 = f_3(X_1, X_2, X_3)$	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$	Use the rule LE to remove self-cycles and transform the input equations
2	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$		Select X_1 as a virtually coloured node to colour the graph, but we meet the sub-cycle (X_2, X_3)
3	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$		Repeat virtual colouring process, by selecting X_2 node as another virtually coloured node. Now $X_3 = g_3(X_1, X_2)$ is a virtually coloured solution
4	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$	$X_2 = g_2(X_1, g_3(X_1, X_2))$	Use the rule RSS to virtually colour the node X_2
5	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, g_3(\ldots))$ $X_3 = g_3(X_1, X_2)$	$X_2 = h_2(X_1)$	Use the rule LE to obtain a virtually coloured solution
6	$X_1 = g_1(X_2, X_3)$ $X_2 = h_2(X_1)$ $X_3 = g_3(X_1, X_2)$	$X_3 = h_3(X_1)$	Then colour the node X_3 by the rule RSS: $X_3 = g_3(X_1, h_2(X_1)) = h_3(X_1)$
7	$X_1 = g_1(X_2, X_3)$ $X_2 = h_2(X_1)$ $X_3 = h_3(X_1)$	$X_1 = g_1(h_2(X_1), h_3(X_1))$	From the virtually coloured results of X_2 and X_3, actually colour X_1 by the rule RSS
8	$X_1 = g_1(\ldots, \ldots)$ $X_2 = h_2(X_1)$ $X_3 = h_3(X_1)$	$X_1 = Y_1$	Apply the rule LE for the equation: $X_1 = g_1(h_2(X_1), h_3(X_1))$
9	$X_1 = Y_1$ $X_2 = h_2(X_1)$ $X_3 = h_3(X_1)$	$X_2 = Y_2$ $X_3 = Y_3$	Apply the rule RSS for the equations $X_2 = h_2(X_1)$ and $X_3 = h_3(X_1)$ to actually colour X_2 and X_3

described is still applicable to remove this form of ***goto*** statement. Provided we can identify the statements that follow a label within a guarded statement, then the statement equation can be easily built. For any branch statement

$$\textbf{\textit{if}}\ C \rightarrow S;\ \textbf{\textit{label}}: S' \,[\,]\neg C \rightarrow S''\ \textbf{\textit{fi}};\ S'''$$

the statement sequence that follows the ***label*** is certainly $S'; S'''$. We can build its corresponding statement equation $X = (S'; S''')\,[|\textbf{\textit{label}}/X, \ldots |]$. For instance, the labels ***label2***, ***label3*** and ***label4*** in Fermat's Algorithm below are guarded. We have used this principle to form their equations.

For those cases where a label occurs in a sequential loop body, i.e.

$$\textbf{\textit{do}}\ G \rightarrow S;\ \textbf{\textit{label}}: S'\ \textbf{\textit{od}};\ S''$$

the statement that follows the label ***label*** is S'; ***do*** $G \rightarrow S$; ***label***$: S'$ ***od***; S''. We can construct its corresponding statement equation as $X = (S';$ ***if*** $G \rightarrow S;$ $X\,[\,]\neg G \rightarrow S''$ ***fi***$)\,[|\textbf{\textit{label}}/X, \ldots |]$ because $S';$ ***do*** $G \rightarrow S;$ ***label***$: S'$ ***od***; S'' is equivalent to $S';$ ***if*** $G \rightarrow S;\ X\,[\,]\neg G \rightarrow S''$ ***fi*** since X represents the statement that follows the label ***label*** to the end. Alternatively we may use the equation $X = S';$ ***do*** $G \rightarrow S;\ S'$ ***od***; $S''[|\textbf{\textit{label}}/X, \ldots |]$. This is just the result obtained after applying LE to the original equation.

Using either of these methods we can always define a statement variable equation for a label which is guarded by complex guarded statements.

For example, with the label in

$$\textbf{\textit{do}}\ G \rightarrow S;\ \textbf{\textit{if}}\ C \rightarrow S_1;\ \textbf{\textit{label}}: S_2\,[\,]\neg C \rightarrow S_3\ \textbf{\textit{fi}};\ S_4\ \textbf{\textit{od}};\ S_5$$

its corresponding statement equation is

$$X = S_2;\ S_4;\ \textbf{\textit{do}}\ G \rightarrow S;\ \textbf{\textit{if}}\ C \rightarrow S_1;\ S_2\,[\,]\neg C \rightarrow S_3\ \textbf{\textit{fi}};\ S_4\ \textbf{\textit{od}};\ S_5[|\textbf{\textit{label}}/X, \ldots |]$$

We can always use this method to remove any internal ***goto*** statements from a loop body or any sequential statement block.

In this section we have presented a strategy for removing ***goto*** statements from any program. It may introduce a number of ***break***s. However, a previously developed process [17] enables us to eliminate these ***break***s and produce an equivalent, transformed program, that is ***goto***-free and ***break***-free.

4. USING THE STATEMENT EQUATION SOLUTIONS TO RECONSTRUCT PROGRAMS

For any equation set containing n statement equations we assume each solution to be Y_i, where $i \in [1, n]$. The solution Y_i gives us the required transformed program segment. The remaining task is to correctly substitute the solution. Since the solution Y_i represents a semantically

equivalent statement sequence starting from the statement labelled by ***label***$_i$ and proceeding to the end, then the following rule may be applied:

Program Reconstruction Rule (PRR)

Given any program/segment S_0; ***label***$_1$: S_1; ***label***$_2$: S_2; ...; ***label***$_n$: S_n where Y_i, $i \in [1, n]$ is the solution to the ith equation, then the equivalent ***goto***-free program/segment is $(S_0$; ***label***$_1$: $S_1)[|$***label***$_1/Y_1$, ***label***$_2/Y_2, \ldots,$ ***label***$_n/Y_n|]$

5. APPLICATION OF THE METHOD

To illustrate the reengineering process we have defined we will apply it to two examples.

Example 1

The first example we will consider involves transforming an *incorrect* implementation of *Fermat's Algorithm* [1] that employs ***goto***s. The original implementation has the form:

$x := 2^*\sqrt{n} + 1$; $y := 1$; $r := (\sqrt{n})^2 - n$;

label *1*: ***if*** $r < 0 \rightarrow$ ***goto label*** *3* ***fi***;

label *2*: $r := r - y$; $y := y + 2$; ***goto label*** *1*;

label *3*: ***if*** $r = 0 \rightarrow$ ***goto label*** *4* ***fi***;
$r := r + x$; $x := x + 2$; ***goto label*** *2*;

label *4*: $u := (x - y)$ div 2

Before building the statement equations, we need to convert the program into a form that corresponds to the standard equations:

label *1*: ***if*** $r < 0 \rightarrow$ ***goto label*** *3*
$[]\, r \geqslant 0 \rightarrow$ ***label*** *2*: $r := r - y$; $y := + 2$; ***goto label*** *1*;
label *3*: ***if*** $r = 0 \rightarrow$ ***goto label*** *4*
$[]\, r \neq 0 \rightarrow r := r + x$;
$x := x + 2$; ***goto label*** *2*;
label *4*: $u := (x - y)$ div 2
fi
fi;

We then can easily build the following statement equation set (see Figure 4).

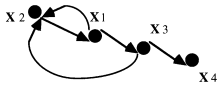

FIGURE 4.

$X_1 =$ ***if*** $r < 0 \rightarrow X_3\, []\, r \geqslant 0 \rightarrow X_2$ ***fi***

$X_2 = r := r - y$; $y := y + 2$; X_1

$X_3 =$ ***if*** $r = 0 \rightarrow X_4 []\, r \neq 0 \rightarrow r := r + x$; $x := x + 2$; X_2 ***fi***

$X_4 = u := (x - y)$ div 2

where $u := (x - y)$ div 2 is already the solution to X_4.

Now we use the colouring algorithm to find solutions. First, we select X_1 as a virtually coloured node. X_2 is then virtually coloured automatically, as is X_3. This process corresponds to the following transformations:

$X_1 =$ ***if*** $r < 0 \rightarrow X_3\, []\, r \geqslant 0 \rightarrow X_2$ ***fi***

$=$ ***if*** $r < 0 \rightarrow$ ***if*** $r = 0 \rightarrow X_4\, []\, r \neq 0 \rightarrow r := r + x$; $x := x + 2$; X_2 ***fi***
$[]\, r \geqslant 0 \rightarrow X_2$
fi (RSS)

$=$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$; $X_2\, []\, r \geqslant 0 \rightarrow X_2$ ***fi*** (since the $r = 0$ branch is unreachable)

$=$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; X_2 (SNS)

$=$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$; X_1 (RSS)

Therefore we actually colour X_1 by a solution (corresponding to a non-terminating loop) ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$ ***od*** according to LE. The solutions for all the equations are:

$Y_1 =$ ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$ ***od***

$Y_2 = r := r - y$; $y := y + 2$; ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$ ***od***

$Y_3 =$ ***if*** $r = 0 \rightarrow Y_4\, []\, r \neq 0 \rightarrow r := r + x$; $x := x + 2$; $r := r - y$; $y := y + 2$; ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow \ldots$ ***od fi***

$=$ ***if*** $r = 0 \rightarrow u := (x - y)$ div 2
$[]\, r \neq 0 \rightarrow x := x + 2$; $y := y + 2$; $r := r + x - y$; ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow \ldots$ ***od***
fi (removing the redundant assignment $r := r + x$)

$Y_4 = u := (x - y)$ div 2

To reconstruct the Fermat's Algorithm above, we just need to substitute Y_1 for the sequence starting with ***label*** *1*: ***if*** $r < 0 \rightarrow \ldots$; ***label*** *4*: $u := (x - y)\,\mathrm{div}\,2$. Hence we end up with the equivalent program:

$x := 2^*\sqrt{n} + 1;\ y := 1;\ r := (\sqrt{n})^2 - n;$

do true $\rightarrow$ ***if*** $r < 0 \rightarrow x := x + 2;\ r := r + x$ ***fi***;

$y := y + 2;\ r := r + x - y$ ***od***

In this example a non-terminating defect is discovered. We should remark that during transformation of the equation X_1, an unreachable path $r = 0 \rightarrow X_4$ is found under its precondition r<0. This indicates the original program *never* reaches the last statement although the syntactic representation of the program contains this statement. The example demonstrates that our method not only eliminates ***goto***s from programs but that it can be used to detect logical defects and to optimize programs.

Example 2:

As another example, consider a Pascal procedure taken from software used in industry (P. Farrow, personal communication).

```
procedure INTI(var TD, DTD, RKO, T, DT,
                   JS, JN, IO, JS4: integer);
  label 1, 2, 3, 4, 5;
  begin
      IO := RKO; JN := 0;
      if IO = 4 then goto 1;
      JS := JS + 1;
      if JS = 3 then JS := 1; if JS = 2 then goto 5;
      DT := DTD;
  3:  TD := TD + DT; T := TD; goto 5;
  1:  JS4 := JS4 + 1; if JS4 = 5 then JS4 := 1;
      if JS4 = 1 then goto 2;
      if JS4 = 3 then goto 4;
      goto 5;
  2:  DT := DTD div 2;
      goto 3;
  4:  TD := TD + DT; DT := 2*DT; T := TD;
  5: end
```

This problem may most conveniently be restructured by treating it as two distinct sub-problems (Figure 5A). The more complicated of the sub-problems begins with the sequence commencing at label 3. We will deal with this sub-problem first, then incorporate it into an overall solution. Before applying our proposed restructuring method, we transform the segment

$\ldots$; ***if*** JS4 = 1 ***then goto*** 2;
if JS4 = 3 ***then goto*** 4; ***goto*** 5; $\ldots$

into the standard form:

$\ldots$; ***if*** JS4 = 1 ***then goto*** 2 ***elsif*** JS4 = 3
then goto 4 ***else goto*** 5; $\ldots$

Mapping the structure into a statement equation set we get:

$X_1 =$ JS4 := JS4 + 1; ***if*** JS4 = 5 ***then*** JS4 := 1;
if JS4 = 1 ***then*** X_2 ***elsif*** JS4 = 3 ***then*** X_4 ***else*** X_5
$X_2 =$ DT := DTD div 2; X_3
$X_3 =$ TD := TD + DT; T := TD; X_5
$X_4 =$ TD := TD + DT; DT := 2^*DT; T := TD; X_5
$X_5 =$ ***skip***

where X_5 has been coloured already (Figure 5B).

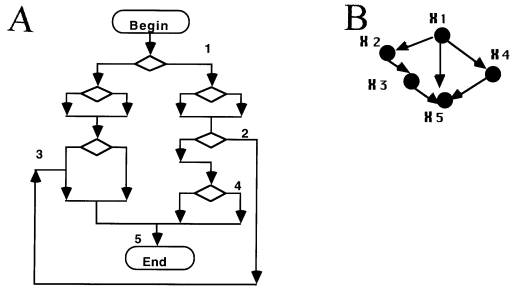

FIGURE 5

The solutions Y_2, Y_3 and Y_4 for X_2, X_3 and X_4 are straightforward to find because there are no cycles involved. We will therefore only show the workings for finding the solution Y_1 for X_1:

$Y_1 =$ JS4 := JS4 + 1; ***if*** JS4 = 5 ***then*** JS4 := 1; ***if*** JS4 = 1 ***then*** Y_2 ***elsif*** JS4 = 3 ***then*** Y_4 ***else*** Y_5

= ***if*** JS4 = 4 ***then*** JS4 := JS4 + 1; JS4 := 1 ***else*** JS4 := JS4 + 1; (SNS)
if JS4 = 1 ***then*** DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 3 ***then*** TD := TD + DT; DT := 2^*DT; T := TD;
else skip (RSS)

= ***if*** JS4 = 4 ***then*** JS4 := 1; ***if*** JS4 = 1 ***then*** DT ... ***elsif*** JS4 = 3 ***then*** TD ... ***else skip***
else JS4 := JS4 + 1; ***if*** JS4 = 1 ***then*** DT ... ***elsif*** JS4 = 3 ***then*** TD ... ***else skip*** (SNS)

= ***if*** JS4 = 4 ***then*** JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 0 ***then*** JS4 := JS4 + 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 2 ***then*** JS4 := JS4 + 1; TD := TD + DT; DT := 2^*DT; T := TD
else JS4 := JS4 + 1 (removing redundancy, SNS)

The solutions for this equation set are therefore:

Y_1 = ***if*** JS4 = 4 ***then*** JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 0 ***then*** JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 2 ***then*** JS4 := 3; TD := TD + DT; DT := 2^*DT; T := TD
else JS4 := JS4 + 1

Y_2 = DT := DTD div 2; TD := TD + DT; T := TD

Y_3 = TD := TD + DT; T := TD

Y_4 = TD := TD + DT; DT := 2^*DT; T := TD

Y_5 = ***skip***

Before substitution of the solutions for this program, we have to transform the original code:

if IO = 4 ***then goto*** 1; JS := JS + 1;
if JS = 3 ***then*** JS := 1;
if JS = 2 ***then goto*** 5; DT := DTD;
3: TD; = TD + DT; . . . . . .

into the equivalent standard form using SNS:

if IO = 4 ***then goto*** 1;
if JS = 2 ***then*** JS := JS + 1; JS := 1;
DT := DTD; (***label***) 3: . . .
elsif JS = 1 ***then*** JS := JS + 1; JS := 2; ***goto*** 5
else JS := JS + 1; DT := DTD; (***label***) 3: . . . ,

and furthermore,

if IO = 4 ***then goto*** 1
elsif JS = 2 ***then*** JS := 1; DT := DTD; (***label***) 3: . . .
(redundant JS := JS + 1 is removed)
elsif JS = 1 ***then*** JS := 2; ***goto*** 5
(redundant JS := JS + 1 is removed)
else JS := JS + 1; DT := DTD; (***label***) 3: . . . ,

We therefore have an equivalent procedure body

IO := RKO; JN := 0;
if IO = 4 → ***if*** JS4 = 4 ***then*** JS4 := 1;
DT := DTD div 2;
TD := TD + DT; T := TD
elsif JS4 = 0 ***then*** JS4 := 1;
DT := DTD div 2;
TD := TD + DT; T := TD
elsif JS4 = 2 ***then*** JS4 := 3;
TD := TD + DT; DT := 2^*DT; T := TD
else JS4 := JS4 + 1
[]IO ≠ 4 ∧ JS = 2 → JS := 1; DT := DTD;
TD := TD + DT; T := TD
[]IO ≠ 4 ∧ JS = 1 → JS := 2
[]IO ≠ 4 ∧ JS ≠ 1 ∧ JS ≠ 2 → DT := DTD;
TD := TD + DT; T := TD
fi

So we end up with the equivalent procedure:

```
procedure INTI(var TD, DTD, RKO, T, DT, JS, JN, IO, JS4: integer);
begin
    IO := RKO; JN := 0;
    if IO = 4 ∧ (JS4 = 0 ∨ JS4 = 4) → JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
    []IO = 4 ∧ JS4 = 2 → JS4 := 3; TD := TD + DT; DT := 2*DT; T := TD
    []IO = 4 ∧ JS4 ≠ 0 ∧ JS4 ≠ 2 ∧ JS4 ≠ 4 → JS4 := JS4 + 1
    []IO ≠ 4 ∧ JS = 1 → JS := 2
    []IO ≠ 4 ∧ JS = 2 → JS := 1; DT := DTD; TD := TD + DT; T := TD
    []IO ≠ 4 ∧ JS ≠ 1 ∧ JS ≠ 2 → JS := JS + 1; DT := DTD; TD := TD + DT; T := TD
    fi
end
```

which can be easily transformed into the following procedure in Pascal:

```
procedure INTI(var TD, DTD, RKO, T, DT, JS, JN, IO, JS4: integer);
begin
    IO := RKO; JN := 0;
    if IO = 4 then if JS4 = 0 ∨ JS4 = 4 then
                        begin JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD end
                        else if JS4 = 2 then
                            begin JS4 := 3; TD := TD + DT; DT := 2*DT; T := TD end
                        else JS4 := JS4 + 1
    else if JS = 1 then JS := 2
        else if JS = 2 then begin JS := 1; DT := DTD; TD := TD + DT; T := TD end
            else begin JS := JS + 1; DT := DTD; TD := TD + DT; T := TD end
end
```

This method is very useful for re-engineering complex structures containing ***goto***s. The transformed program is more readable, easier to maintain and more reliable. A number of quality defects, such as redundancy, and non-termination can be removed or detected, and other tasks, such as verification and derivation of a specification from programs, may also be achieved.

4. CONCLUSION

We have introduced a process, that enables the elimination of all ***goto*** statements from any program. The advantage of this approach over other alternatives that have been proposed is that it is securely based on formal semantics. To ensure the generality of this process it has been necessary to formulate a general and powerful formally based method to remove ***goto*** statements from any program structure. The processes we have developed can be used to detect and/or remove a number of quality and reliability defects from programs.

This process specifies a set of general, language-independent, widely applicable, formal, but also practical, techniques for improving the quality of programs. It can render difficult code systematically manageable without the usual tedium of pouring over existing complex program structures. The recommended strategy is first to tame such structures using the methods we have suggested and, only then, to proceed to analyse the code for its intent. The method is quite straightforward to apply manually and it has the potential for automated implementation.

The fact that the key process employed is very similar to that used for solving sets of algebraic equations should make the method attractive to a wide audience. An important use of the method is as a preprocessing step for the more general re-engineering processes: loop rationalization and loop normalization [17, 18]. Another interesting application of the method is to use it directly to eliminate recursion in programs. We can do this because any recursion can be replaced by a mechanism involving ***goto*** statements.

REFERENCES

[1] Dromey, R. G. and McGettrick, A. D. (1992) On Specifying Software Quality. *Software Quality J.*, **1**, 43–74.

[2] Ashcroft, E. and Manna, Z. (1972) The Translation of GOTO Programs to WHILE Programs. In *Proc. IFIP Congr. 71*, pp. 250–255, North-Holland, Amsterdam.

[3] Gray, A. S. and Whitty, R. W. (1982) Correspondence of Flowchart Schemata. *Comp. J.*, **25**, 495.

[4] Knuth, D. E. (1974) Structured Programming with GOTO statements. *ACM Computing Surveys*, **6**, 261–301.

[5] Knuth, D. E. and Floyd, R. W. (1971) Notes on Avoiding GOTO Statements. *Inf. Proc. Letters*, **1**, 23–31.

[6] Oulsnam, G. (1982) Unravelling Unstructured Programs. *Comp. J.*, **25**, 379–387.

[7] Williams, M. H. (1976) Generating Structured Flow Diagrams: the Nature of Unstructuredness. *Comp. J.*, **20**, 45–50.

[8] Williams, M. H. (1982) A Comment on the Decomposition of Flowchart Schemata. *Comp. J.*, **25**, 393–396.

[9] Williams, M. H. (1983) Flowchart Schemata and the Problem of Nomenclature. *Comp. J.*, **26**, 270–276.

[10] Williams, M. H. and Chen, G. (1985) Restructuring Pascal Programs Containing GOTO Statements. *Comp. J.*, **28**, 134–137.

[11] Williams, M. H. and Ossher, H. L. (1978) Conversion of Unstructured Flow Diagrams to Structured Form. *Comp. J.*, **21**, 161–167.

[12] Carpenter, B. E., Doran, R. W. and Hopper, K. (1977) Non-recursive Recursion. *Softw. Pract. Exp.*, **7**, 263–268.

[13] Goldschlager, L. M. (1981) Recursion in Small Storage. *Softw. Pract. Exp.* **11**, 745–751.

[14] Back, R. J. R. (1988) A Calculus of Refinements for Program Derivation. *Acta Informatica*, **25**, 593–625.

[15] Dijkstra, E. W. and Scholten, C. S. (1989) *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin.

[16] Pan, S. (1995) *Software Quality Improvement, Specification Derivation and Quality Measurement Using Formal Methods*. Ph.D. Thesis, Griffith University, Australia.
[17] Pan, S. and Dromey, R. G. (1996) Reengineering Loops. *Comp. J.*, **39**, 184–202.
[18] Pan, S. and Dromey, R. G. (1993) *Loop Normalization*. Research Report, Griffith University, Australia.
[19] Alagic, S. and Arbib, M. (1978) *The Design of Well-structured and Correct Programs*. Springer-Verlag, New York.
[20] Farrow, P. (1993) Private communication. Centre of Information Technology Research, University of Queensland, Australia.