Re-engineering Loops

SI PAN AND R. GEOFF DROMEY

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

Loops with multiple-exits and flags 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 systematic, generally applicable procedure, called *loop rationalization*, which removes such features and logically simplifies loop structures is presented. This method, which is founded on the *principle of separation of concerns*, employs strongest postcondition calculations and congruent equivalence transformations to improve loops. A by-product of the process is that it detects a range of defects such as unreachable code and a class of non-termination problems.

Received March 29, 1994; revised March 20, 1996

1. INTRODUCTION

In imperative programs, loops are usually regarded as the most difficult structures to implement, analyse, reuse, modify and prove correct. We contend that the use of multiple exits and flags exacerbates the difficulties of dealing with loops. Such features significantly detract from the structural integrity, simplicity, reliability and the ultimate quality of programs.

We suggest that the best way to combat the difficulties associated with loops is to keep their structures as simple and as direct as possible. Most programmers would probably agree with this design philosophy. The problem is, however, that there is no powerful, widely applicable, systematic means for achieving this design goal or judging when it has been realized. This situation has done much to inhibit our ability to produce consistently high-quality imperative programs.

An examination of the literature in this area over the past two decades reveals that there have been a number of studies [1–18] on methods for improving the structural quality of loops. Much of this work has focused on removing *goto*s and exception handling, and developing structures that support multiple conditions for loop termination. Assessing these results we conclude that:

- those that involve transformations heavily rely on pattern matching and therefore lack generality;
- they sometimes result in transformations that change the invariant and/or post-termination properties of the original loop structure;
- they offer language-specific rather than generally applicable transformations and improvements;
- they introduce logical and textual inefficiencies that make no positive contribution to the goal of realizing loop structures that are as simple and direct as possible.

To overcome the limitations of existing methods we propose a formally based, systematic technique, called *loop rationalization.* It may be used to simplify single and nested loop structures that contain multiple exits and various forms of logical redundancy. The method also removes/detects a number of other quality defects associated with loops.

More specifically the improvements and simplifications induced by loop rationalization are:

- all multiple exits are removed from loop bodies,
- post-termination structure is removed from loop bodies,
- flags used for termination are removed,
- unreachable branches are detected,
- static logical redundancy is removed from loop guards and the loop body.

What loop rationalization offers is a formal, practical and rigorous method for re-engineering existing loops into new improved loop structures that satisfy their original specification. Rationalized loops employ only direct guards and possess a single point of entry and exit. What is more, their structures conform to a corresponding graph theoretic ideal associated with strong connectedness. An important feature of the loop rationalization process is that it is easy to apply manually and it is amenable to automation. Loop rationalization obviates the need to pour over complex and difficult-to-understand loop structures peppered with flags, logical redundancy and multiple exits. Instead we can systematically apply loop rationalization to understand the essence of, and improve, such delinquent program structures.

Before discussing the formal basis of loop rationalization we will illustrate informally key steps of the method and the sort of results that may be obtained. Consider the following loop, taken from an uncommented commercial C program:

```
......
while ((Op = getopt(arge, argv, Vop)) ! = ERROR) {
    if ((char) Op == Opq) {
        QFlag = TRUE;
        if (LFlag) {
            Error = TRUE;
            break; }
        if (SFlag) {
            Error = TRUE;
            break; }
    }
    else { if ((char) Op == Opl) {
        LFlag = TRUE;
        if (QFlag) {
            Error = TRUE;
            break; }
        LevelStr = optarg;
        }
        else {if ((char) Op == Ops) {
            SFlag = TRUE;
            Subsystem = optarg;
            if (QFlag) {
                Error = TRUE;
                break; }
            }
        else { Error = TRUE;
                break; }
        }
    }
}
```

The first major step in the process of loop rationalization is to carry out a transformation that, in operational terms, effectively delays as far as possible the execution of all state-changing assignments. That is, if it is possible to delay an assignment until some additional condition has been established then this should always be done. This transformation uncovers the strongest precondition under which each assignment may execute. Imposing this requirement also effectively collapses complex branch structures within a loop to a simple uniform form. Using guarded commands [19] we call this structure the *multiply branched statement* (MBS) form, i.e:

do G → ***if*** $C_1 \rightarrow S_1$ [] $C_2 \rightarrow S_2$ [] . . . . [] $C_n \rightarrow S_n$ ***fi od***

A loop that has undergone transformation to the MBS form establishes the same postcondition as the original form. Applying this transformation for the loop above we establish that it has a MBS structure with eight branches, i.e

```
    do (Op = getopt(arge, argv, Vop)) ! = ERROR →
{B1}if (Op == Opq)∧LFlag → QFlag = TRUE; Error = TRUE; break
{B2}[] (Op == Opq)∧¬LFlag∧SFlag → QFlag = TRUE; Error = TRUE; break
{B3}[] (Op == Opq)∧¬LFlag∧¬SFlag → QFlag = TRUE
{B4}[] (Op == Opl)∧ QFlag → LFlag = TRUE; Error = TRUE; break
{B5}[] (Op == Opl)∧¬QFlag → LFlag = TRUE; LevelStr = optarg
{B6}[] (Op == Ops)∧QFlag → SFlag = TRUE; Subsystem = optarg; Error = TRUE; break
{B7}[] (Op == Ops)∧¬QFlag → SFlag = TRUE; Subsystem = optarg
{B8}[] (Op ! = Opq)∧(Op ! = Opl)∧(Op ! = Ops) → Error = TRUE; break
    fi
od
```

We may use this *precondition-resolved* loop body to study systematically the execution behaviour of the loop as it passes from one iteration to the next. For example, after execution of B_2, the loop terminates because of the ***break*** statement; whereas

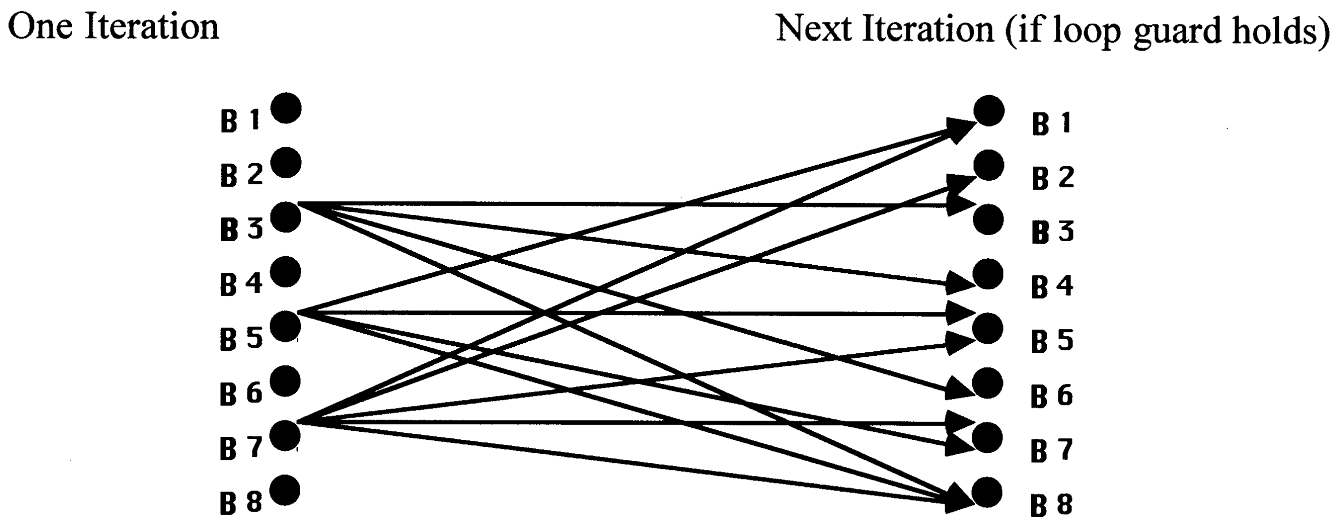

FIGURE 1. Figure shows which branches are reachable from each branch as a loop passes from one iteration to the next.

after execution of B_3, the loop may either terminate directly if (Op = getopt(arge, argv, Vop)) == ERROR holds, or in the next iteration it can enter only one of B_3, B_4, B_6 or B_8. This is because execution of B_3 establishes the condition ¬LFlag∧¬SFlag∧QFlag which *implies the negation of* the preconditions for executing branches other than B_3, B_4, B_6 or B_8. Examining which branches may be reached in the next iteration from each branch yields the bipartite graph shown in Figure 1.

A primary goal of this paper is to show how to calculate this information formally. We may use this information to construct the *branch successor graph* (BSG) for the loop (see Figure 2).

The BSG may be used to both assess and improve the structural quality of programs [20]. Important logical (semantic) and structural characteristics of a loop may be conveniently interpreted in terms of features of the BSG. *Well-structured* loops have a BSG where each node (corresponding to a branch in the MBS loop body) connects directly to all other nodes. *This identifies the requirement where for each branch of the loop, after execution of that branch it should be possible to access all other branches in the next iteration.* If this situation does not prevail there exist redundancies in the loop guard and/or branch guards. For instance, our example shows that only three branches B_3, B_5 and B_7 have successors whereas the other branches have none. This means that if the control-flow enters any of the branches B_1, B_2, B_4, B_6 or B_8 it will terminate after execution of the branch. Removing the terminating branches we are left with the following loop structure:

```
do (Op = getopt(arge, argv, Vop) == Opq)∧¬LFlag∧¬Flag →QFlag = TRUE
[] (Op = getopt(...) == Opl)∧¬QFlag → LFlag = TRUE; LevelStr = optarg
[] (Op = getopt(...) == Ops)∧QFlag → SFlag = TRUE; Subsystem = optarg
od;
if (Op == Opq)∧(LFlag∨SFlag) → QFlag = TRUE; Error = TRUE
[] (Op == Opl)∧QFlag →LFlag = TRUE; Error = TRUE
[] (Op == Ops)∧QFlag →SFlag = TRUE; Subsystem = optarg; Error = TRUE
[] (Op ! = ERROR)∧(Op ! = Opq)∧(Op ! = Opl)∧(Op ! = Ops) → Error = TRUE
fi
```

This transformed loop achieves the same result as the original loop. It also removes much of the redundancy from the original loop since a number of components of the body of the original loop have been relegated to post-termination processing. The resulting loop has only three branches compared with the eight in the original. Also the improved loop has only

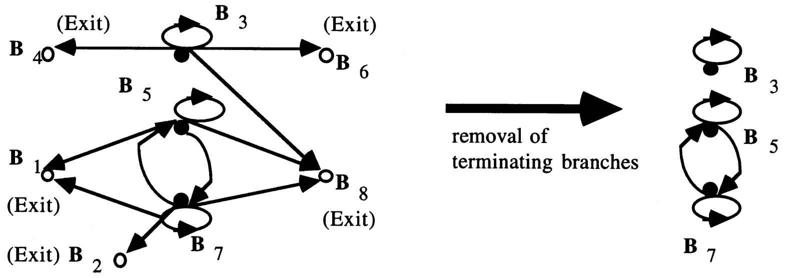

FIGURE 2. Branch successor graph before and after removal of terminating branches.

a single exit. Later we will see how the loop consisting of branches B_3, B_5 and B_7 may be decomposed into two loops corresponding to the two strongly connected components (B_3, B_5 and B_7).

Recapping, the major steps in the loop rationalization process are:

1. Transform the loop to the MBS form.
2. Calculate strongest postconditions for all branches of the MBS structure.
3. Construct the BSG to guide loop reengineering (GLR).
4. Remove all terminating branches using MEL or DRG transformations.

We are now ready to examine the formal aspects of loop rationalization.

2. TRANSFORMATIONS FOR DERIVING THE MULTIPLY BRANCHED STATEMENT FORM

Before it is possible to apply the transformations that directly realize the MBS form for a loop body, it is necessary to apply some preliminary transformations to accommodate the three types of exit/termination statement (other than the loop guard) that are commonly used in conventional programming languages to bring about loop termination. The statements we must deal with and their actions are as follows:

- ***goto*** jumps from the loop body to a specified labelled statement;
- ***break*** terminates iteration then executes the statement directly following the loop;
- ***return*** transfers control directly out of the function/procedure that encapsulates it.

2.1. Preliminary transformations

We have previously developed a formal process [16] to eliminate all ***goto***s from any program. Here, we will therefore focus only upon removal of ***break*** and ***return*** statements.

After removal of ***goto***s from a loop body only ***break***s, ***return***s and *the statements that make the loop guard false* are left to cause multiple-exits. Generally, the last kind of ‘exit’ [4] may not be detected by pure syntactic analysis, because it is often difficult to neatly capture in a powerful pattern description all possible forms that a particular kind of statement might assume. *This is an important reason for applying formal semantic calculations to identify exits and indirect termination mechanisms for loops.*

What we intend to show is that it is possible to remove all forms of exit and indirect termination from loops. The result we obtain is a loop structure that is terminated solely by its guard(s). The process employs a bottom-up (internal-to-external loop) strategy that allows us to consider, at each stage of the process, only simple, single MELs.

In what follows we assume any ***break*** or ***return*** is guarded by a branch statement. If it is not, it must be guarded by a loop or occur in a sequential block directly. These latter forms, which represent inappropriate structural compositon, can be directly removed from a loop by the following transformation rules:

do G → S; ***break***; ... ***od*** $\models$ ***if*** G → ***Sfi***[1]
do G → S; ***return***; ... ***od*** $\models$ ***if*** G → S; ***return*** **fi**

In the second rule we keep the ***return*** statement within the guarded statement, because this inappropriate loop may be nested inside a number of other loops. Another reason for keeping this ***return*** is because it may assign a value or state to a variable (as can happen in the C language).

After removal of ***goto***s and unguarded ***break***s and ***return***s, loops with multiple-exits must be sequences containing guarded statements. Such branched sequences are in a form where it is possible to apply the equivalence transformations that enable us to transform a loop body into the MBS form. We will now consider the process and the required transformations.

2.2. The MBS transformations

A number of equivalence tranformations are needed to transform a loop to the MBS form. Their role is to *absorb* into a branch statement those state-changing statements (assignments and I/O statements) that either precede or follow the branch statement. For this purpose, the following equivalence transformations may be employed:

- x := E; ***if*** C → S_1 [] ¬C → S_2 ***fi*** $\models$ ***if*** C[E/x] → x := E; S_1 [] ¬C[E/x] → x := E; S_2 ***fi***
- ***read***(x); ***if*** C → S_1 [] ¬C → S_2 ***fi*** $\models$ ***if*** C → ***read***(x); S_1 [] ¬C → ***read***(x); S_2 ***fi*** when $x \notin \mathbf{V}(C)$
- ***write***(E); ***if*** C → S_1 [] ¬C → S_2 ***fi*** $\models$ ***if*** C → ***write***(E); S_1 [] ¬C → ***write***(E); S_2 ***fi***
- ***if*** C → S_1 [] ¬C → S_2 ***fi***; S $\models$ ***if*** C → S_1; S [] ¬C → S_2; S ***fi***

[1] This is an augmentation of Dijkstra‘s *Guarded Commands*, which is semantically equivalent to ***if*** G ***then*** S ***endif*** in imperative languages.

where C[E/x] indicates that all occurrences of x in C are replaced by E and $x \notin V(C)$ indicates that x is not a member of the set of variables present in C.

These rules, when applied to exhaustion, ensure that the transformed sequences are branched and guarded as early and as deeply as possible. They enable us to optimize and remove redundancies from statement sequences.

Examples:

1. x := y+2; ***if*** y>z → z := x [] y ⩽ ˙z → z := y ***fi*** ⊨ ***if*** y>z → x := y+2; z := x [] y ⩽ ˙z → x := y+2; z := y ***fi***
   x := y+2; ***if*** x>z → z := x [] x ⩽ z → z := y ***fi*** ⊨ ***if*** y+2>z → x := y+2; z := x [] y+2 ⩽ ˙z → x := y+2; z := y ***fi***
2. ***read*** (x); ***if*** y>z → z := x [] y ⩽ z → z := y ***fi*** ⊨ ***if*** y>z → ***read***(x); z := x [] y ⩽ z → ***read***(x); z := y ***fi***

However, we cannot use the transformation rules above to absorb the following ***read*** statement:

read(x); ***if*** x>z → z := x [] x ⩽ z → z := y ***fi***

because the ***read***-variable x is used by the branch guards x>z and x ⩽ z.

After applying these transformations, we end up with loop bodies that fit into one of the following categories:

- A *MBS* of the form:

$$\textbf{\textit{if}}\ C_1 \rightarrow S_1\ []\ C_2 \rightarrow S_2\ []\ ...\ []\ C_n \rightarrow S_n\ \textbf{\textit{fi}}$$

where $\forall i \forall j((i,j \in [1,n] \wedge j \neq i) \Rightarrow (C_i \Rightarrow \neg C_j))$ and $C_1 \vee C_2 \vee \ldots \vee C_n \equiv \textbf{true}$. We refer to these two conditions as the *MBS conditions*.

- A *bounded MBS* of the form:

$$S;\ \textbf{\textit{if}}\ C_1 \rightarrow S_1\ []\ C_2 \rightarrow S_2\ []\ ...\ []\ C_n \rightarrow S_n\ \textbf{\textit{fi}}$$

where the *MBS conditions* apply.

Weakest precondition calculations underpin these various equivalence transformations, for example.

$$S;\ \textbf{\textit{if}}\ C \rightarrow S_1\ []\ \neg C \rightarrow S_2\ \textbf{\textit{fi}}\ |\!=\ \textbf{\textit{if}}\ wp(S, C) \rightarrow S; S_1\ []\ \neg wp(S, C) \rightarrow S; S_2\ \textbf{\textit{fi}}$$

This being the case, we must be prepared to deal with situations where weakest precondition calculations are not easily realizable. For instance, the weakest precondition calculation wp(S, C) returns *undefined* for arbitrary input data when S is a ***read***(x) and the branch guard C also involves the variable **x**. It may also be difficult to calculate wp(S, C) when S is a loop structure or a procedure that changes at least one variable in C. Both these structural situations prevent us from moving all statements that precede a branch statement into that statement‘s branches. Hence it is necessary to recognize and handle appropriately the bounded MBS form. Once we have a loop in the MBS form we may use strongest postcondition calculations to help identify inefficiences and defects in the loop’s logical structure.

3. STRONGEST POSTCONDITION CALCULATIONS

Strongest postcondition calculations provide a powerful means for guiding the restructuring of loops of the form ***do*** G → . . .; ***if*** B → S . . . ***od*** containing one or more guarded branches S. The task is to identify the conditions that apply after the program statement sequence S executes under a precondition P. The strongest postcondition for this is denoted by sp(P, S). For some of the implementations that we will sketch, a variant of the *guarded commands* notation will be used. Annotating the program S we have:

{ P }	- precondition
S	- program statement sequence
{ sp(P, S) }	- strongest postcondition established by S

The theory supporting strongest postconditions calculations has many parallels with that for weakest preconditions [19, 21]. Previous studies [19] characterize the semantics, i.e. the strongest postconditions, of the different statements in an imperative language, in terms of executing each statement type under a given precondition, P:

DS0:	sp(P∨R, S)	≡ sp(P, S) ∨ sp(R, S)
DS1:	sp(P, n := E)	≡ (n = E)∧∃n:P
DS2:	sp(P, n := E(n))	$\equiv P[E^{-1}(n)/n]$
DS3:	sp(P, ***if*** C → S1 [] ¬C → S2 ***fi***)	$\equiv sp(P \wedge C, S_1) \vee sp(P \wedge \neg C, S_2)$
DS4:	sp(P, ***do*** G → S ***od***)	$\equiv \neg G\ (\exists i{:}\ 0 \leqslant i{:}\ sp(P, (\textbf{\textit{if}}\ G \rightarrow S\ \textbf{\textit{fi}})^i))$
DS5:	sp(P, S1;S2)	≡ sp(sp(P, S1), S2)

Examples

$\mathrm{sp}(i>-1,\ i := i+2) \equiv i-2>-1 \equiv i>1$ (DS2)

$\mathrm{sp}(x = y+1,\ z := y+c) \equiv (x = y+1) \wedge (z = y+c)$ (DS1)

Although the results above give, in principal, a calculation method for the strongest postcondition, they are limited for a number of reasons. First, the term $\exists n$: P (DS1) in the strongest postcondition for assignment is not explicitly indicated since it stipulates only that the variable n is bounded by the precondition P. Also the inverse function $E^{-1}(x)$ in DS2 may be difficult to compute or $E^{-1}(x)$ may even be a relation. These properties make direct calculation difficult. Secondly, in DS4 for the loop, the depth i of the calculation is not fixed. This again makes direct calculation of strongest postconditions for loops difficult.

To overcome these theoretical barriers, we have developed an extended computational model to calculate strongest accessible postconditions for all statements from a given precondition. This model includes calculations for loops [21]. It provides a theoretically sound and implementable basis for improving existing software. For full details of such calculations and applications it is necessary to refer to other more specialized reports [20, 21]. Here we highlight the treatments.

Those only interested in the practical aspects of loop rationalization may, in the first instance at least, wish to skip the theoretical aspects of the discussion in the rest of Section 3. As our example in the introduction suggests, in most cases, in practice, working out the strongest postcondition for a branch is relatively straightforward and possible to achieve without having a deep understanding of strongest postcondition theory. Our primary goal is to make loop rationalization accessible and directly useful to the average programmer who has little interest in formal methods. It is, however, important to put this work on a firm theoretical footing. In what follows we will briefly examine the key theoretical and practical issues associated with strongest postcondition calculations. For those solely interested in applying loop rationalization a careful study of the examples and main transformations should be all that is needed to understand and usefully apply the method in practice.

3.1. Calculations for assignment

To overcome the difficulties with strongest postconditions calculations for assignments in practical programs we must augment the original theory [19]. We may prove

$\mathrm{sp}(P,\ x := E) \equiv P^{V-\{x\}} \wedge x = E$,

where $P^{V-\{x\}}$ is defined as follows [19]:

when $P \Rightarrow (x = e)$: $P^{V-\{x\}} \equiv P[e/x]$ (R1)

otherwise: $P^{V-\{x\}} \equiv P[y/x]$, where y is a fresh variable (R2)

$\mathrm{sp}(P,\ x := E(x)) \equiv [E_1^{-1}(x)/x] \vee P[E_2^{-1}(x)/x] \equiv P[E_n^{-1}(x)/x]$,

where $E_1^{-1}(x)$, $E_2^{-1}(x)$, ... and $E_n^{-1}(x)$ are all inverse functions of $x = E(x)$

Example:

$$
\begin{aligned}
\mathrm{sp}(a[i]>a[j],\ x := a[i];\ a[i] := a[j];\ a[j] := x) &\equiv \mathrm{sp}(\mathrm{sp}(a[i]>a[j]^{V-\{x\}} \wedge x = a[i],\ a[i] := a[j]),\ a[j] := x) && \text{(DS5)}\\
&\equiv \mathrm{sp}(\mathrm{sp}(a[i]>a[j] \wedge x = a[i],\ a[i] := a[j]),\ a[j] := x) && \text{(R2)}\\
&\equiv \mathrm{sp}((a[i]>a[j] \wedge x = a[i])^{V-\{a[i]\}} \wedge a[i] = a[j],\ a[j] := x) &&\\
&\equiv \mathrm{sp}((a[i]>a[j])[x/a[i]] \wedge a[i] = a[j],\ a[j] := x) && \text{(R1)}\\
&\equiv (x>a[j] \wedge a[i] = a[j])^{V-\{a[j]\}} \wedge a[j] = x &&\\
&\equiv x>a[i] \wedge a[j] = x && \text{(R1)}
\end{aligned}
$$

In practice, these theoretical formulas may still make the calculations difficult. Firstly $P(x)^{V-\{x\}}$ may not always be easily obtained from P(x). Secondly, the inverse functions of x = E(x) may be undefined (e.g. $x = x/0$), difficult to evaluate (e.g. $x = Ax^{25}+Bx$) or infinite (e.g. $x = x\ m\mathbf{od}\ 2$). However, from a theoretical viewpoint, for any assignment x := e, $\mathrm{sp}(P,\ x := e) \equiv \mathrm{sp}(P,\ t := x;\ x := e[t/x])^{V-\{t\}}$, where t is a fresh variable and sp(P, t := x; x := e[t/x]) can always be calculated. Since the following calculation/evaluation never uses the auxiliary variable t again, we need not remove t from sp(P, t := x; x := e[t/x]), because it does not affect any subsequent calculation/evaluation. This auxiliary variable gives us a practical and simple approach to calculating strongest postconditions involving assignments. More importantly, these calculation difficulties reflect the presence of quality defects in a program where a variable is being used for two or more purposes. The treatment indicates consistency rules for the use of variables in programs. Further details are reported in [17].

3.2. Calculations for I/O, break and return statements

We need to include strongest postcondition calculations for I/O, ***break*** and ***return*** statements in order to apply this formal approach to imperative programs. The I/O statements we use here are *denoted* as ***read***(x) and ***write***(E), where x and E are a declared variable and an expression, respectively. The declared variable x may be input by a user. Since static semantics cannot handle dynamic input, we assume any ***read***-statement ***read***(x) is independent, i.e. x is an arbitrary datum in its type domain.

The strongest postconditions for I/O statements are:

sp(P, ***read***(x)) $\equiv P^{V-\{x\}} \wedge$ **read**(x),

sp(P, ***write***(E)) $\equiv P \wedge$ **write**(E)

The statement ***read***(x) corresponds to a non-deterministic assignment to x with an arbitrary value of the appropriate type. The predicate r**ead**(x) can be treated as a total function on the type domain $\boldsymbol{D}_T(x)$ of x, i.e. $\forall x(x \in \boldsymbol{D}_T(x) \Rightarrow \mathbf{read}(x)) \wedge \forall x(x \notin \boldsymbol{D}_T(x) \neg \mathbf{read}(x))$. Certainly any ***read***-statement may fail to terminate at run-time (e.g. due to a type-error). However, such dynamic errors are beyond the scope of this work. The predicate **write**(E) requires that all variables in E are initialized and type-matched.

Similarly, the strongest postconditions for ***break*** and ***return*** statements are:

sp(P, ***break***) $P \wedge$ **break**,

sp(P, ***return***) $\equiv P \wedge$ **return**

where **break** and **return** are predicates. The statement ***return*** may return a value/status and terminate the whole function/ process. We therefore have sp(P$\wedge$**return**, S) $\equiv$ P for any statement sequence S that follows such a ***return*** statement and is bounded by the current block (function, procedure or program, etc.). The statement ***break*** forces termination of a loop by transferring control to the statement that directly follows the loop. We therefore have sp(P$\wedge$break, S) $\equiv$P for any statement sequence S in the current loop body that follows such a ***break*** statement. For instance, given a loop ***do*** G $\rightarrow$ ...; ***if*** C $\rightarrow$ ...; ***break***; S_1 [] $\neg$C $\rightarrow$ ... ***fi***; S_2 **od** and a precondition P at the execution point before the ***break*** statement, the statement sequence following the ***break*** statement in the current loop body is S_1; S_2 and we have sp(P$\wedge$**break**, S_1; S_2) $\equiv$ P.

3.3. Calculations for loops

Direct calculations of strongest postconditions for loops using DS4 is not practical because the depth of the calculations is not fixed. In order to overcome the difficulty, we introduce a process to calculate a closed weaker form called the *strongest accessible conjunctive postcondition* sp_{ac}(P, ***do*** G $\rightarrow$ S **od**). This is bounded and strong enough for improving programs and deriving specifications from implementable programs.

To perform the calculations for loops we assume that the strongest accessible conjunctive postcondition after the first iteration, Q_1, consists of m+1 predicates in the form $R_1 \wedge R_2 \wedge \ldots \wedge R_m \wedge G'$, and, after the second iteration, Q_2 will contain m+1 predicates $R'_1, R'_2, \ldots R'_m, G''$. When S does not contain any redundancy, each R'_i is called the *Update Predicate* of R_i. We treat $R'' \wedge R'$ as the update predicate of R', where R'' is the predicate satisfying $sp_{ac}(R', S) \Rightarrow R''$ and $R' \equiv wp(S, R'')$. Generally, such an R' is created by each iteration, either from assignments, branch conditions or guards or subloops. For example, $a[i-1] = i-1 \wedge a[i] = i$ in Q_2 is the update predicate of $a[i] = i$ in Q_1 for the loop i := 0;***do*** i<N $\rightarrow$ i := i+1; a[i] := i ***od***. Update predicates capture the invariant form of the loop. Given a predicate R and its update predicate R', we may use a function $g(\mathbf{x})$ to describe $R'(\mathbf{x})$ by $R(g(\mathbf{x}))$ and furthermore in the following iteration, $\mathbf{x}$ will still follow the way $g(\mathbf{x})$ changes. Generally, after the *f*-th iteration, the update predicate of $R(\mathbf{x})$ should be $R(g^0(\mathbf{x}))$ where $(g^0(\mathbf{x}) = \mathbf{x})$. We can use the following rules to obtain the strongest conjunctive result from $R(\mathbf{x}) \vee R(g(\mathbf{x})) \vee \ldots \vee R(g^{f-1}(\mathbf{x})))$ where *f* is the iterative function:

$R(\mathbf{x})@R(g(\mathbf{x})) \equiv \exists j \in [1, f] R(g^{j-1}(\mathbf{x}))$ when $\neg(R(g(\mathbf{x})) \Rightarrow R(\mathbf{x}))$ SF1

$R(\mathbf{x})@R(g(\mathbf{x})) \; \forall j \in [1, f] R(g^{j-1}(\mathbf{x})) \equiv R(g^{f-1}(\mathbf{x}))$ when $R(g(\mathbf{x})) \Rightarrow R(\mathbf{x})$ SF2

These formulas together with others [21] assist us to formally extract the strongest invariant properties from a loop. SF2 yields the strongest invariant component $\forall j \in [1,i] \; a[j] = j$ for the example above.

3.4. Calculations for procedures/functions

In practice, we also need to extend strongest postcondition calculations to handle procedures/functions. There are two cases we need consider. A procedure/function without side-effects can be treated as straightforward as a normal statement/expression (where the local variables should be removed from the strongest postcondition). It becomes more complex when we need to calculate strongest postconditions for procedures/function with side-effects.

Given a precondition P for a procedure Q with side-effects, the strongest postcondition should be sp(P, Q). During calculation, the state-changing statements (i.e. assignments and ***read***-statements) in Q change the states of variables in P. As a result, the final variable states in sp(P, Q) may be different from the states in P.

For the case of functions that are used as operands in expressions the following situation applies. When a function has a side-effect, it may change some states of variables in the expression. For example, consider an expression x+*f*(x)+x*x, where the side-effect function f(x) increases x by 1 then returns the current value of x. When the initial value of x is 1, the expression may correspond to either 1+2+2*2 if x*x is evaluated after f(x) or 1+2+1*1 if x*x is evaluated before f(x). The value of the expression depends on the evaluation-order that the selected compiler chooses. In order to model the evaluation-order problem, we denote an expression E containing a side-effect function f(...) as E(x, f(...),y), where the variable sets x and y are evaluated before and after execution of f(...) respectively. Given any precondition P for such an E(x, f(...),y), the strongest postcondition is defined as:

$\mathrm{sp}(P, E(x, f(...),y)) \equiv \mathrm{sp}(P, x' := x; f' := f(...); E(x', f', y))^{V-\{x', f'\}}$ where x′ and f' are fresh variables (sets)

This formula is modelled by the following steps:
1. Store x using a fresh variable set x′ and a multiple assignment, i.e. $\mathrm{sp}(P, x' := x)$;
2. Evaluate $E(x', f(...), y)$, i.e. $\mathrm{sp}(\mathrm{sp}(P, x' := x), f' := f(...); E(x', f', y))$;
3. Remove the auxiliary variables, i.e. $\mathrm{sp}(P, x' := x; f' := f(...); E(x', f', y))^{V-\{x', f'\}}$

Example

$$\begin{aligned}
\mathrm{sp}(x = 1 \wedge y > x,\ t := x + f(x) + x*x) &\equiv \mathrm{sp}(x = 1 \wedge y > x,\ x' := x;\ f' := f(x);\ t := x' + f' + x*x)^{V-\{x',\ f'\}} \\
&\qquad \text{where } f(x)\text{: } x := x+1;\ \mathbf{return}(x) \\
&\equiv \mathrm{sp}(x = 1 \wedge y > x-1 \wedge x' = x,\ f' := f(x);\ t := x' + f' + x*x)^{V-\{x',\ f'\}} \quad \text{(DS5, DS1)} \\
&\equiv \mathrm{sp}(x-1 = 1 \wedge y > x-1 \wedge x' = x-1 \wedge f' = x,\ t := x' + f' + x*x)^{V-\{x',\ f'\}} \quad \text{(DS5, DS2)} \\
&\equiv \mathrm{sp}(x = 2 \wedge y > x-1 \wedge x' = 1 \wedge f' = x \wedge t = x' + f' + x*x)^{V-\{x',\ f'\}} \quad \text{(DS1)} \\
&\equiv x = 2 \wedge y > x-1 \wedge t = 1+2+2*2
\end{aligned}$$

4. BRANCH SUCCESSOR GRAPH CONSTRUCTION

BSGs may be constructed from the results of strongest postcondition calculations for MBS loop bodies. They allow important logical (semantic) and structural characteristics of a loop to be conveniently interpreted in terms of graphs. Furthermore, meaningful macroscopic transformations on the BSG directly correspond to a set of structural improvements on the program. In carrying out the *structural refinement* of loops the ultimate goal is to transform the BSG into a single or a set of strongly connected components. The corresponding loop structures exhibit no static logical redundancy and therefore conform to a logical ideal or normal form.

To construct a BSG for a loop structure we must calculate the strongest postcondition for the execution of each branch in the n-branch loop structure. The next step is then to establish for each branch, by use of logical implication, which of the other branches are accessible to that branch. Once this has been done the BSG can be constructed.

Given a loop with a branched body of the form

$$\textbf{\textit{do}}\ G \rightarrow S;\ \textbf{\textit{if}}\ C_1 \rightarrow S_1\ []\ C_2 \rightarrow S_2\ []\ ...\ []\ C_n \rightarrow S_n\ \textbf{\textit{fi}}\ \textit{od},$$

where the MBS conditions apply, the n postconditions for each (the *i*-th) of the branches are of the form $\mathrm{sp}(\mathrm{sp}(G, S) \wedge C_i, S_i)$. These postconditions determine the successor set of each branch, that is, $\beta_i = \{1,2,..., n\} - \alpha_i$, where $\alpha_i = \{j | \mathrm{sp}(\mathrm{sp}(G, S) \wedge C_i, S_i) \Rightarrow \neg G \text{ or } \mathrm{sp}(\mathrm{sp}(\mathrm{sp}(G, S) \wedge C_i, S_i) \wedge G), S) \Rightarrow \neg C_j\}$. Each branch $C_i \rightarrow S_i$ maps to a separate node of a directed graph. All the nodes that β_i indicates are then connected. This means, there is a directed arc from i to j if and only if $j \in \beta_i$. These nodes and edges form the *BSG*. An example is shown in Section 1 for the commercial C loop.

Any *exit conditions* satisfy $\mathrm{sp}(\mathrm{sp}(G \wedge P, S) \wedge C_i, S_i) \Rightarrow \neg G$, for $i \in [1,n]$. If the *i*-th branch causes an exit, then after execution of its body, a condition is established which implies that the guard G for the loop is false (that is, a condition for termination of the loop has been established). This allows us to define the *branch exit set:*

Definition: branch exit set
Given any loop ***do*** G → ; ***if*** $C_1 \rightarrow S_1\ []\ C_2 \rightarrow S_2\ []\ ...\ []\ C_n \rightarrow S_n$ ***fi od*** under the precondition P, the *branch exit set* γ_{exit} *is defined as*
$\gamma_{exit} = \{i \mid sp(sp(G \wedge P, S) \wedge C_i, S_i) \Rightarrow \neg G$ *or* S_i *contains a* ***break*** or ***return*** and $\mathrm{sp}(G \wedge P, S) \wedge C_i \neq \mathbf{false}$, for $i \in [1,n]\}$
where the MBS conditions apply.

The branch exit set describes the execution paths that, after execution once, terminate the loop, i.e. those execution paths which have no successors. We should remark the condition $\mathrm{sp}(G \wedge P, S) \wedge C_i = \mathbf{false}$ is used for extracting branches that are unreachable during the first iteration (they may be reachable in subsequent iterations). This definition is based purely on formal calculations. It allows us to handle various forms of exit from loops. Based on the definition of γ_{exit}, we can define a MEL as follows:

Definition: *multiple – exit loop (MEL)*
Any loop is called a *MEL* if it has a non-empty branch exit set γ_{exit}.

This formal definition for a MEL captures not only exits using ***break***s and ***return***s but also other forms [4], such as:

do exec $=$ true $\rightarrow$...; ***if*** C $\rightarrow$...; exec := **false** [] $\neg$ C $\rightarrow$... ***fi od*** and

do status $=$ 'iterating' $\rightarrow$...; ***if*** C $\rightarrow$ ...; status := 'terminating' [] $\neg$ C $\rightarrow$... ***fi od***.

5. FORMAL TRANSFORMATIONS FOR REMOVING TERMINATING BRANCHES

Once a loop with multiple exits has been transformed into an MBS structure the task that remains is to remove all the exiting branches that have been identified by the exit set. To do this the following device is used. We may denote any MEL by ***do*** G $\rightarrow$ S; ***if*** $C_e \rightarrow S_e$ [] C¬e $\rightarrow$ S¬e ***fi*** *od*, where the composite 'branch' $C_e \rightarrow S_e$ collects together all branches that the exit set indicates. Certainly the branch guards C_e and $C_{\neg e}$ satisfy $C_e \Rightarrow C_{\neg e}$ and $C_e \wedge C_{\neg e} \equiv$**true**. The ultimate goal for improvement of a MEL is to remove the branch $C_e \rightarrow S_e$ from the transformed loop.

5.1. Transformation for multiple-exit loops with multiple-branched bodies

When S $=$ *skip* (the empty statement), the loop body S; ***if*** $C_e \rightarrow S_e$ [] $C_{\neg e} \rightarrow S_{\neg e}$ ***fi*** is just a MBS ***if*** $C_e \rightarrow S_e$ [] $C_{\neg e} \rightarrow S_{\neg e}$ ***fi***. The equivalence transformation rule for removal of multiple-exits for this case is:

Rule for MELs
do G $\rightarrow$ ***if*** $C_e \rightarrow S_e$ [] $C_{\neg e} \rightarrow S_{\neg e}$ ***fi*** *od* $| =$ ***do*** $G \wedge C_{\neg e} \rightarrow S_{\neg e}$ ***od***; ***if*** G $\rightarrow S_e$ ***fi*** where $sp(sp(G, S) \wedge C_e, S_e) \Rightarrow \neg G$

This rule enables us to remove all multiple-exits (corresponding to S_e) from the original loop, and form a rationalized loop that contains only non-exiting branches (corresponding to $S_{\neg e}$). This rule is also applicable for loops where the branch guards C_e and $C_{\neg e}$ have side-effects. The exits are detected by calculation under the weakest precondition G of the loop's body. This means that for any precondition P, even if C_e has a side-effect these detected exits also terminate the loop because $sp(sp(P \wedge G, S) \wedge C_e, S_e) \Rightarrow sp(sp(G, S) \wedge C_e, S_e) \Rightarrow \neg G$. When, however, G involves a side-effect, we should change the original guard of the last statement ***if*** G $\rightarrow S_e$ ***fi*** into its corresponding guard that has no side-effect. This is necessary because the rationalized loop guard $G \wedge C_{\neg e}$ is executed *before* ***if*** G $\rightarrow S_e$ ***fi***. Later in Section 6.1, we will show this situation.

When $S_{\neg e}$ is a MBS of the form ***if*** $C_1 \rightarrow S_1$ [] ... [] $C_t \rightarrow S_t$ ***fi*** and $C_{\neg e} \equiv C_1 \vee \ldots \vee C_t$, the *recommended* structure of the rationalized loop ***do*** $G \wedge C_{\neg e} \rightarrow S_{\neg e}$ ***od*** is

do $G \wedge C_1 \rightarrow S_1$ [] $G \wedge C_2 \rightarrow S_2$ [] ... [] $G \wedge C_t \rightarrow S_t$ ***od***

This removes redundant testing from the guard $G \wedge C_{\neg e}$ and the branch guards in $S_{\neg e}$. The redundant testing shows up in the original rationalized loop ***do*** $G \wedge C_{\neg e} \rightarrow S_{\neg e}$ ***od*** as follows:

do $G \wedge (C_1 \vee \ldots \vee C_t) \rightarrow$ ***if*** $C_1 \rightarrow S_1$ [] $C_2 \rightarrow S_2$ [] ... [] $C_t \rightarrow S_t$ ***fi od***.

The *recommended* loop structure may be very simply and efficiently translated into imperative languages like C, etc using the following structure,

do G $\rightarrow$ ***if*** $C_1 \rightarrow S_1$ ***elsif*** $C_2 \rightarrow S_2$ ***elsif*** ... ***elsif*** $C_t \rightarrow S_t$ ***else break od***.

5.2. Transformations for MELs with bounded multiple-branched bodies

Different transformations are needed to deal with bounded multiple-branch loop bodies depending on whether or not the guard G is what we call dynamically redundant. Strongest postcondition calculations may be used to detect the two cases.

<u>Case I</u> - Dynamically redundant guard (DRG) present
For any loop with a bounded body ***do*** G $\rightarrow$ S; ***if*** $C_e \rightarrow S_e$ [] $C_{\neg e} \rightarrow S_{\neg e}$ ***fi od***, previous results [20] show that when $sp(sp(G, S) \wedge C_{\neg e}, S_{\neg e}) \Rightarrow G$, the corresponding branch, by its execution, never has a chance to bring about termination. In this instance, the guard G is said to be dynamically redundant and we may accordingly transform the loop to an equivalent form using the following rule:

Dynamically redundant guard
do G $\rightarrow$ S; ***if*** $C_e \rightarrow S_e$ [] $C_{\neg e} \rightarrow S_{\neg e}$ ***fi od*** $| =$ ***if*** G $\rightarrow$ S; ***do*** $C_{\neg e} \rightarrow S_{\neg e}$; S ***od***; S_e ***fi***
when $sp(sp(G, S) \wedge C_e, S_e) \Rightarrow \neg G$ and $sp(sp(G, S) \wedge C_{\neg e}, S_{\neg e}) \Rightarrow G$.

The transformation rule DRG enables us to remove all multiple-exits and form a rationalized loop that contains only non-exiting branches for loops whose bodies are in the form of S; ***if*** $C_e \rightarrow S_e$ [] $C_{\neg e} \rightarrow S_{\neg e}$ ***fi***. In a similar way to the MEL transformation, the DRG rule is applicable for cases where the branch guards C_e and $C_{\neg e}$ have side-effects. Unlike for the MEL rule, the resulting segment is a branch statement whose loop guard is weakened to $C_{\neg e}$.

When sp(sp(G, S)$\wedge$C$_e$, S$_e$)$\Rightarrow$ $\neg$G and $\neg$(sp(sp(G, S)$\wedge$C$_{\neg e}$, S$_{\neg e}$)$\Rightarrow$G) certainly $\neg$(sp(sp(G, S)$\wedge$C$_{\neg e}$, S$_{\neg e}$)$\Rightarrow$ $\neg$G, otherwise the second branch also belongs to the exit set. In this situation we cannot find a logically simpler structure in the guarded command language [20] than ***do*** G $\rightarrow$ S; ***if*** C$_e$ $\rightarrow$ S$_e$ [] C$_{\neg e}$ $\rightarrow$ S$_{\neg e}$ ***fi od*** itself. In a language like C, with its side-effects, we may, however, simplify this type of structure. For example,

```
do { getc(char);                       |= getc(char);
   if (char == EndofLine) {break }        if (char ! = EndofLine) {printf(char)};
   else {printf(char)}                    while (getc(char) ! = EndofLine){printf(char)}
}while ()
```

After removing textual redundancy [21], we end up with

while (***getc***(char) ! = EndofLine){***printf***(char)}

However, this method eventually introduces a side-effect into the guard. We can absorb S in ***do*** G $\rightarrow$ S; ***if*** C$_e$ $\rightarrow$ S$_e$ [] C$_{\neg e}$ $\rightarrow$ S$_{\neg e}$ ***fi od*** into a new guard using a Boolean function GS: ***if*** G $\rightarrow$ S; GS := true [] $\neg$G $\rightarrow$ GS := false ***fi***, then apply MEL to obtain the equivalent loop ***do*** GS $\rightarrow$ ***if*** C$_e$ $\rightarrow$ S$_e$ [] C$_{\neg e}$ $\rightarrow$ S$_{\neg e}$ ***fi od***. This is not generally useful because the guard contains a side-effect.

Case II - Guard G is not dynamically redundant

For any ***do*** G $\rightarrow$ S; ***if*** C$_e$ $\rightarrow$ S$_e$ [] C$_{\neg e}$ $\rightarrow$ S$_{\neg e}$ ***fi od***, when sp(sp(G, S)$\wedge$C$_e$, S$_e$)$\Rightarrow$ $\neg$G and $\neg$(sp(sp(G, S)$\wedge$C$_{\neg e}$, S$_{\neg e}$)$\Rightarrow$G), that is, when G is not dynamically redundant, the following two approaches may be applied to remove multiple-exits from the loop.

1. *Formal approach*

A formal approach to removing multiple-exits from ***do*** G $\rightarrow$ S; ***if*** C$_e$ $\rightarrow$ S$_e$ [] C$_{\neg e}$ $\rightarrow$ S$_{\neg e}$ ***fi od***, is based on formal semantics. This means the rationalized result has the same specification as the original loop.

Although for S; ***if*** C$_e$ $\rightarrow$ S$_e$ [] C$_{\neg e}$ $\rightarrow$ S$_{\neg e}$ ***fi*** the statement S cannot be absorbed into the MBS, the loop guard G may be used to prefix S$_{\neg e}$. This results in

```
do G → S; if Ce → Se [] C¬e → S¬e fi od
|=do G → S; if Ce → Se [] C¬e∧wp(S¬e, G) → S¬e [] C¬e∧¬wp∧(S¬e, G) → S¬e fi od
|=if G → S; do C¬e∧wp(S¬e, G) → S¬e; S od; if Ce → Se [] C¬e → S¬e fi fi        (DRG)
```

because sp(sp(G, S)$\wedge$C$_{\neg e}$$\wedge$wp(S$_{\neg e}$, G), S$_{\neg e}$) $\Rightarrow$sp(wp(S$_{\neg e}$, G), S$_{\neg e}$) $\Rightarrow$G and sp(sp(G, S)$\wedge$C$_{\neg e}$$\wedge\neg$wp(S$_{\neg e}$, G), S$_{\neg e}$) $\Rightarrow$sp($\neg$wp(S$_{\neg e}$, G), S$_{\neg e}$) $\Rightarrow\neg$G. Later we will show how this approach may be used for Atkinson's example (see Section 6.3).

2. *Informal approach*

For any loop ***do*** G $\rightarrow$ S; ***if*** C$_e$ $\rightarrow$ S$_e$ [] C$_{\neg e}$ $\rightarrow$ S$_{\neg e}$ ***fi od***, when C$_e$ and G cannot be promoted into S and S$_{\neg e}$, respectively, there must exist a ***read***(s) or procedure(s) or loop(s) in both S and S$_{\neg e}$. For instance, ***do*** G(x,y) $\rightarrow$ ***read***(x); ***if*** C(x) $\rightarrow$ S$_1$; ***break*** [] $\neg$C(x) $\rightarrow$ ***read***(y); S$_2$ ***fi od*** is a typical example. From the viewpoint of software quality [21, 22] this sort of loop has an inconsistent (inhomogeneous) invariant which should always be avoided. Because the exiting branch C$_e$ $\rightarrow$ S$_e$ is executed only once, it should not be in the loop structure. For such structures, reconstruction should retain the essential parts S and S$_{\neg e}$ in the loop, and remove S$_e$.

To handle MELs with these properties it is best to produce a rationalized loop with a consistent invariant that no longer satisfies the original specification. Practically, we should strive to achieve the form ***do*** G$'$ $\rightarrow$ S; S$_{\neg e}$ ***od***; S$_e$. For example, an appropriate structure for the loop ***do*** G(x,y) $\rightarrow$ ***read***(x); ***if*** C(x) $\rightarrow$ S$_1$; ***break*** [] $\neg$C(x) $\rightarrow$ ***read***(y); S$_2$ ***fi od*** is ***do*** G$'$(x,y) $\rightarrow$ ***read***(x); ***read***(y); S$_2$ ***od***; S$_1$. This changes the original specification but in a way that is rational and appropriate.

6. EXAMPLES

Several examples will now be used to illustrate the practicality of loop rationalization based on the process and the transformations we have formulated. In many instances the process can be conducted in a largely informal, but still rigorous way. Experience has shown that it is relatively easy to teach to people how to apply the process even if they do not fully understand all aspects of its formal basis.

6.1. Single loop

The first example we will consider involves the fragment of a commercial C program given in Section 1. The result of transforming the original loop to MBS form has already been given in Section 1. The next stage involves identifying and removing the terminating branches. The strongest postcondition calculations needed here for the different branches under each resolved precondition are straightforward. Results from these calculations under the precondition true for all branches are

TABLE 1.

Branch	*Resolved branch precondition*	*Branch postcondition*	*Successor*
1	((Op = get...) == Opq)∧LFlag	(Op == Opq)∧LFlag∧QFlag∧Error∧***break***	{}
2	((Op = get...) == Opq)∧¬LFlag∧ SFlag	(Op == Opq)∧¬LFlag∧SFlag∧QFlag∧Error∧***break***	{}
3	((Op = get...) == Opq)∧¬LFlag ∧¬SFlag	(Op == Opq)∧¬LFlag∧¬SFlag∧QFlag	{3, 4, 6, 8}
4	((Op = get...) == Opl)∧QFlag	(Op == Opl)∧QFlag∧LFlag∧Error∧***break***	{}
5	((Op = get...) == Opl)∧¬QFlag	(Op == Opl)∧¬QFlag∧LFlag∧LevelStr = optarg	{1, 5, 7, 8}
6	((Op = get...) == Ops) ∧ QFlag	(Op == Ops)∧ QFlag ∧ SFlag ∧ Subs... = optarg∧ Error ∧ ***break***	{}
7	((Op = get...) == Ops)∧¬QFlag	(Op == Ops)∧¬QFlag∧SFlag∧ Subsystem = optarg	{1, 2, 5, 7, 8}
8	((Op = get...)! = ERROR)∧(Op! = Opq)∧(Op ! = Opl)∧(Op ! = Ops)	(Op! = ERROR)∧(Op! = Opq)∧(Op ! = Opl)∧(Op! = Ops)∧ Error∧***break***	{}

summarized in Table 1. From the table we may deduce that the exit set is $\gamma_{\text{exit}} = \{1,2,4,6,8\}$, i.e. the branches that contain loop exiting ***break*** statements.

The original loop guard (Op = getopt(...)! = ERROR) contains a side-effect. According to the MEL transformation the branch guard that follows the rationalized loop should be (Op! = ERROR). To remove the terminating branches {1,2,4,6,8}, we need to apply MEL, and logically simplify the resulting guards from (Op! = ERROR)∧(Op == Opq) into (Op == Opq), and so on. The rationalized result was shown in Section 1. Note that the two terminating branches that set QFlag and Error may be combined. Carrying out these steps we end up with a transformed algorithm that can be easily implemented in a deterministic form in the C language, i.e.

```
do {Op = getopt(arge, argv, Vop);
   if (Op == Opq) && (!LFlag) && (!SFlag) { QFlag = TRUE }
   elsif (Op == Opl) && (!QFlag) { LFlag = TRUE; LevelStr = optarg }
   elsif (Op == Ops) && (!QFlag) { SFlag = TRUE; Subsystem = optarg }
   else { break }
}while ();
if (Op == Opq) && ((LFlag)||(SFlag)){ QFlag = TRUE; Error = TRUE; }
elsif (Op == Opl) && (QFlag) { LFlag = TRUE; Error = TRUE; }
elsif (Op == Ops) && (QFlag) { SFlag = TRUE; Subsystem = optarg; Error = TRUE; }
elsif (Op ! = ERROR) && (Op ! = Opq) && (... ! = Opl) && (... ! = Ops) { Error = TRUE; }
```

The process of removing the terminating branches corresponds to, in graph theoretic terms, removing all the leaves from the graph (see Figure 1).

Compared with the original unstructured loop, this transformed loop is much easier to understand. Only those elements of the original loop that are essential are retained in the loop body. All non-guard-terminated exits (caused by ***break***s) which do not belong to the loop, are removed from the rationalized loop. The resulting control structure is significantly clearer and simpler because it contains no *static logical redundancy* [21, 22]. However, this transformed loop segment still contains some *dynamic logical redundancy* [21, 22] which is signalled by the fact that all three branches in the transformed BSG, B_3, B_5 and B_7 are not strongly connected—they in fact indicate the need for a loop with two branches and a loop with a single branch if the dynamic redundancy is to be removed. The process of loop normalization [20] can be directly applied to this transformed program in order to remove the remaining dynamic redundancy. It yields the following implementation:

```
Op = getopt(arge, argv, Vop);
if (Op == Opq) && (!LFlag) && (!SFlag) {
   QFlag = TRUE;
   while ((Op = getopt(arge, argv, Vop)) = Opq {};
   if (Op == Opl) { LFlag = TRUE; Error = TRUE; }
   elsif (Op == Ops) { SFlag = TRUE; Subsystem = optarg; Error = TRUE; }
   elsif (Op ! = ERROR) && (Op ! = Opq)&& (Op ! = Opl)&& (Op ! = Ops))
                                                              { Error = TRUE; }
   }
elsif ((Op == Opl)||(Op == Ops)) && (!QFlag) {
   do { if (Op == Opl) { LFlag = TRUE; LevelStr = optarg; }
      elsif (Op == Ops) { SFlag = TRUE; Subsystem = optarg; }
      Op = getopt(arge, argv, Vop);
```

```
  }while ((Op == Opl)||(Op == Ops));
  if (Op == Opq) { QFlag = TRUE; Error = TRUE; }
  elsif (Op ! = ERROR) && (Op ! = Opq)&& (Op ! = Opl)&& (Op ! = Ops))
                                                        {Error = TRUE; }
  }
```

Elsewhere [23] we have shown how a property of the BSG provides an excellent metric for assessing the quality of branched loop structures. The ideal for a multiple-branched loop structure is that in passing from one iteration to the next all branches should be accessible. This means that in the BSG there should be an edge from each node (branch) to every other node. Our commercial C loop example has eight branches (each corresponding to a node). Its ideal BSG should therefore have 8*8 = 64 edges. What we find is that its BSG only has 13 edges or a reachability of $T_d = 13/64$. This low ratio for the reachability gives a strong indication that the loop body has been poorly composed. By comparison the final re-engineered loop has the ideal reachability ratio of 1. *We therefore suggest that as an indicator of branch structure quality this semantic-based index is far more useful than the McCabe Number [24] because it indicates an ideal and how the structure might be improved to realize the ideal.* We have shown elsewhere how BSG have an important role to play in developing a theory of normal forms for program structures [17].

6.2. Nested loop structures

To remove multiple exits from nested loop structures the process of loop rationalization must be applied in a bottom-up fashion. The process begins with the inner-most loop and proceeds progressively towards the outer-most loop. During transformations on an inner loop where multiple exits are removed from the inner loop they must then be dealt with for the enclosing loop to which they have been propagated. To illustrate how the overall process is applied we will transform the following nested multiple-exit C loop structure which has been taken from the literature [10]:

```
NoReply = TRUE;
do { if (SendMessage(RequestMsg)<0)) { printf(‘‘Bad send\n’’); return(RPC-FAILED); };
  while (NoReply)
     if ((status = ReceiveMessage(ReplyMsg,10*Seconds))<0) {
        if (status == TIMEOUT) {
           printf(‘‘Timeout\n’’);
           if (- -RetryCount == 0)
              { printf(‘‘Retry count expired\n’’); return (RPC-FAILED); }
           else {break; } }
        else { printf(‘‘Bad receive. Rea...%d\n’’,status); return(RPC-FAILED); } }
     else { if (InReplyTo(ReplyMsg) == MessageId(RequestMsg)) NoReply = FALSE;
           else { printf(‘‘Bad reply received\n’’); } }
} while (NoReply);
```

The first step is to apply the MBS transformation for the inner loop (note the non-exiting branch is shown below completely in **bold**):

```
while (NoReply) {
  if ((status = Re...Me...(...))<0)∧(status == TIMEOUT)∧(- -RetryCount == 0)
     { printf(‘‘Timeout\n’’); printf(‘‘Retry count expired\n’’); return (RPC-FAILED); }
  [] ((status = Re...Me...(...))<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
     { printf(‘‘Timeout\n’’); break; }
  [] ((status = Re...Me...(...))<0)∧(status ! = TIMEOUT)
     { printf(‘‘Bad receive. Rea...%d\n’’,status); return(RPC-FAILED); }
  [] ((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
     { printf(‘‘Bad reply received\n’’); }
  [] ((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) == MessageId(...))
     { NoReply = FALSE; }
  fi;
  }
```

We may then directly apply the MEL transformation to obtain a much simpler inner loop from which the four terminating branches have been removed, i.e:

```
while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
    { printf(‘‘Bad reply received\n’’); }

if (status<0)∧(status == TIMEOUT)∧(- -RetryCount == 0)
    { printf(‘‘Timeout\n’’); printf(‘‘Retry count expired\n’’); return (RPC-FAILED); }
[] (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
    { printf(‘‘Timeout\n’’); }
[] (status<0)∧(status ! = TIMEOUT)
    { printf(‘‘Bad receive. Rea...%d\n’’,status); return(RPC-FAILED); }
[] (status ⩾ 0)
    { NoReply = FALSE; }
fi;
```

The ***break*** statement in the second branch has been removed after rationalization because it terminates only the inner loop. All ***return*** statements are retained because they also terminate the outer loop. The nested loop then has the form:

```
NoReply = TRUE;
do { if (SendMessage(RequestMsg)<0) { printf(‘‘Bad send\n’’); return(RPC-FAILED); };
    while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
    { printf(‘‘Bad reply received\n’’); }
    if (status<0)∧(status == TIMEOUT)∧(- -RetryCount == 0)
        { printf(‘‘Timeout\n’’); printf(‘‘Retry count expired\n’’); return (RPC-FAILED); }
    [] (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
        { printf(‘‘Timeout\n’’); }
    [] (status<0)∧(status ! = TIMEOUT)
        { printf(‘‘Bad receive. Rea...%d\n’’,status); return(RPC-FAILED); }
    [] (status ⩾ 0)
        { NoReply = FALSE; }
    fi;
} while (NoReply);
```

The next step is to apply the MBS transformation for the body of the outer loop. Unfortunately the nested ***while*** loop is difficult to move into the body of the ***if*** statement. However, the second branch (status<0)∧ (status == TIMEOUT)∧ (–RetryCount ! = 0) { ***printf***(‘‘Timeout\n’’); } maintains the loop guard NoReply indicating the presence of a dynamically redundant guard. Furthermore the other three branches directly terminate the iteration. We may therefore apply the DRG transformation for the outer loop to obtain:

```
NoReply = TRUE;
if (SendMessage(RequestMsg)<0) { printf(‘‘Bad send\n’’); return(RPC-FAILED); }
while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
        { printf(‘‘Bad reply received\n’’); }
while (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
        { printf(‘‘Timeout\n’’);
        if (SendMessage(RequestMsg)<0)
            { printf(‘‘Bad send\n’’); return(RPC-FAILED); }
        while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
            { printf(‘‘Bad reply received\n’’); }
        }
if (status<0)∧(status == TIMEOUT)
    { printf(‘‘Timeout\n’’); printf(‘‘Retry count expired\n’’); return (RPC-FAILED); }
[] (status<0)∧(status ! = TIMEOUT)
    { printf(‘‘Bad receive. Rea...%d\n’’,status); return(RPC-FAILED); }
[] (status ⩾ 0)
    { NoReply = FALSE; }
fi
```

We now process the resulting loop again using the MBS transformation to obtain:

```
while (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
    { if (SendMessage(RequestMsg)<0)
            { printf(‘‘Timeout\n’’); printf(‘‘Bad send\n’’); return(RPC-FAILED); }
        [] (SendMessage(RequestMsg)°0)
            { printf(‘‘Timeout\n’’);
                while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = M...(...))
                    { printf(‘‘Bad reply received\n’’); }
            }
        fi
    }
```

And, after applying the MEL transformation we get:

```
while (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)∧(SendMessage(...Msg) ⩾ 0)
{ printf(‘‘Timeout\n’’);
    while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
            { printf(‘‘Bad reply received\n’’); }
    }
if (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
    { printf(‘‘Timeout\n’’); printf(‘‘Bad send\n’’); return(RPC-FAILED); }
```

The complete rationalized loop then has the form:

```
NoReply = TRUE;
if (SendMessage(RequestMsg)<0) { printf(‘‘Bad send\n’’); return(RPC-FAILED); }
while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
    { printf(‘‘Bad reply received\n’’); }
while (status<0)(status == TIMEOUT)∧(- -RetryCount ! = 0)∧(SendMessage(...Msg) ⩾ 0)
    { printf(‘‘Timeout\n’’);
    while NoReply∧((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
            { printf(‘‘Bad reply received\n’’); }
    }
if (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
    { printf(‘‘Timeout\n’’); printf(‘‘Bad send\n’’); return(RPC-FAILED); }
[] (status<0)∧(status == TIMEOUT)
    { printf(‘‘Timeout\n’’); printf(‘‘Retry count expired\n’’); return (RPC-FAILED); }
[] (status<0)∧(status ! = TIMEOUT)
    { printf(‘‘Bad receive. Rea...%d\n’’,status); return(RPC-FAILED); }
[] (status ⩾ 0)
    { NoReply = FALSE; }
fi
```

For this example, we have not only removed all multiple-exits from the loop but also found the redundant subguard NoReply (which can be removed directly from the loop). The flag NoReply can in fact be completely removed because the precondition establishes NoReply and the loop does not reassign it until the last iteration. This yields the re-engineered implementation which is structurally much simpler than the original:

```
if (SendMessage(RequestMsg)<0) { printf(‘‘Bad send\n’’); return(RPC-FAILED); }
while ((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
        { printf(‘‘Bad reply received\n’’); }
while (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)∧(SendMessage(...Msg) ⩾ 0)
    { printf(‘‘Timeout\n’’);
        while ((status = Re...Me...(...)) ⩾ 0)∧(InReplyTo(...) ! = MessageId(...))
        { printf(‘‘Bad reply received\n’’); }
    }
if (status<0)∧(status == TIMEOUT)∧(- -RetryCount ! = 0)
    { printf(‘‘Timeout\n’’); printf(‘‘Bad send\n’’); return(RPC-FAILED); }
[] (status<0)∧(status == TIMEOUT)
    { printf(‘‘Timeout\n’’); printf(‘‘Retry count expired\n’’); return (RPC-FAILED); }
```

```
[] (status<0)∧(status ! = TIMEOUT)
    { printf(‘‘Bad receive. Rea...%d\n’’,status); return(RPC-FAILED); }
fi
```

We may note in this solution that the main loop ***while*** (status<0)∧(status == TIMEOUT)∧ . . . is directly preceded by and contains within its body another loop, ***while*** ((status = Re...Me...Iterative structures of this form can be most cleanly and simply expressed using the ‘*loop* statement’ found in Ada and some other languages. It consists of an unguarded loop with a single exit in the middle. Using this approach instead of a control structure of the form

A; ***while*** G ***do*** B;A ***end***

we get an implementation of the form

loop A; ***if*** G ***then*** B ***endloop***

where the structure ‘A’ corresponds to the inner loop.

6.3. Removal of guard-terminating branches

We now consider an example (in a Pascal-like language) proposed by Atkinson [4] that was put forward as a strategy for handling MELs [7, 15, 18, 25]. According to our earlier definition the proposed solution is still a MEL. It has the following form:

```
......
PROCEDURE Evaluatef(x: real; VAR f:real; VAR zerodiv:boolean);
......
PROCEDURE Evaluatefdashed(x: real; VAR f:real; VAR zerodiv:boolean);
......
itcount := 0; state := iterating;
REPEAT
    IF abs(oldx) ⩽ assumedzero THEN state := xtoonearzero
    ELSE BEGIN
        Evaluatef(x, fx, zerodivattempted);
        IF zerodivattempted THEN state := zerodivinf
        ELSE BEGIN
            Evaluatefdashed(x, fd, zerodivdashedattempted);
            IF zerodivdashedattempted THEN state := zerodivinfdashed
            ELSE IF abs(fd) ⩽ assumdzero THEN state := flatspotmet
                ELSE BEGIN itcount := itcount+1; oldx := x; x := x-fx/fd;
                    IF abs((x−oldx)/oldx) ⩽ tolerance THEN state := converged
                    ELSE IF itcount = maxits THEN state := maxitsreached
                END
            END
        END
UNTIL state ≠ iterating;
CASE state OF
    converged: ...;
    maxitsreached: ...;
    flatspotmet: ...;
    zerodivinf : ...;
    zerodivinfdashed: ...;
    xtoonearzero: ...
END {case}
```

In this example, since the precondition implies *state = iterating* we can easily transform the *REPEAT* loop into a ***WHILE*** loop. The exit for the first branch can then be handled directly by a MEL transformation. This yields:

```
......
itcount := 0; state := iterating;
WHILE (state = iterating)∧(abs(oldx)>assumedzero) DO
    BEGIN
        Evaluatef(x, fx, zerodivattempted);
        IF zerodivattempted THEN state := zerodivinf
        ELSE BEGIN
            Evaluatefdashed(x, fd, zerodivdashedattempted);
            IF zerodivdashedattempted THEN state := zerodivinfdashed
            ELSE IF abs(fd) $\leqslant$ assumdzero THEN state := flatspotmet
                ELSE BEGIN itcount := itcount+1; oldx := x; x := x-fx/fd;
                    IF abs((x-oldx)/oldx) $\leqslant$ tolerance THEN state := converged
                    ELSE IF itcount = maxits THEN state := maxitsreached
                END
            END
    END;
IF state = iterating THEN state := xtoonearzero;
CASE
    state = converged: ...; /* converged */
    state = maxitsreached: ...; /* maxitsreached */
    state = flatspotmet: ...; /* flatspotmet */
    state = zerodivinf: ...; /* zerodivinf */
    state = zerodivinfdashed: ...; /* zerodivinfdashed */
    state = xtoonearzero: ...; /* xtoonearzero */
END {case}
```

We will now revert to an equivalent version of the algorithm in which the loop is expressed using guarded commands. At the same time we apply the transformations necessary to convert the loop body to the MBS form, i.e:

```
do (state = iterating)$\wedge$(abs(oldx)>assumedzero) $\rightarrow$
    Evaluatef(x, fx, zerodivattempted);
    if zerodivattempted $\rightarrow$ state := zerodivinf
    [] $\neg$zerodivattempted $\rightarrow$
        Evaluatefdashed(x, fd, zerodivdashedattempted);
        if zerodivdashedattempted $\rightarrow$ state := zerodivinfdashed
        [] $\neg$zerodivdashedattempted$\wedge$ abs(fd) $\leqslant$ assumdzero $\rightarrow$ state := flatspotmet
        [] $\neg$zerodivdashedattempted $\wedge$abs(fd)>assumdzero$\wedge$abs(x*fx/fd) $\leqslant$ tolerance $\rightarrow$
            itcount := itcount+1; oldx := x; x := x-fx/fd; state := converged
        [] $\neg$zerodivdashedattempted$\wedge$abs(fd)>assumdzero$\wedge$abs(x*fx/fd)$\wedge$tolerance$\wedge$
            itcount-1 = maxits $\rightarrow$
                itcount := itcount+1; oldx := x; x := x-fx/fd; state := maxitsreached
        [] $\neg$zerodivdashedattempted$\wedge$abs(fd)>assumdzero$\wedge$abs(x*fx/fd)$\wedge$tolerance$\wedge$
            itcount-1$\neq$ maxits $\rightarrow$ itcount := itcount+1; oldx := x; x := x-fx/fd
        fi
    fi
od
```

Reasonable progress is made with applying the MBS transformations until we encounter the two procedures. They block our efforts to remove the multiple-exits associated with the use of the flag *state*. There is, however, an inconsistency in this loop structure. Both procedures share the same input *x* but while the output *fx* is always produced during iteration, the output *fd* is not (this leads to an inhomogeneous loop invariant defect [6]). This discrepancy in the relationship between *fx* and *fd* arises because of the exit branch *zerodivattempted* → *state* := *zerodivinf*. It is necessary to remove this inconsistency, by grouping the two procedure calls together before proceeding with the restructuring of the loop. When we do this we get the following bounded MBS form for the loop:

```
do (state = iterating)$\wedge$(abs(oldx)>assumedzero) $\rightarrow$
    Evaluatef(x, fx, zerodivattempted);
    Evaluatefdashed(x, fd, zerodivdashedattempted);
    if zerodivattempted $\rightarrow$ state := zerodivinf
    [] $\neg$zerodivattempted$\wedge$zerodivdashedattempted $\rightarrow$ state := zerodivinfdashed
    [] $\neg$zerodivattempted$\wedge\neg$zerodivdashedattempted$\wedge$abs(fd) $\leqslant$ assumdzero $\rightarrow$ state := ...
    [] $\neg$zerodivattempted$\wedge\neg$zero...$\wedge$abs(fd)>assumdzero$\wedge$abs(x*fx/fd) $\leqslant$ tolerance $\rightarrow$
            itcount := itcount+1; oldx := x; x := x-fx/fd; state := converged
    [] $\neg$zerodivattempted$\wedge\neg$zero...$\wedge$abs(fd)>assumdzero$\wedge$abs(x*fx/fd)>tolerance$\rightarrow$
        itcount-1 = maxits $\rightarrow$ itcount := itcount+1; oldx := x; x := x-fx/fd; state := maxitsreached
    [] ¬zerodivattempted∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd)>tolerance→
       itcount-1=maxits → itcount := itcount+1; oldx := x; x := x-fx/fd
    fi
  od
```

When we calculate wp(itcount := itcount+1; oldx := x; x := x-fx/fd, abs(oldx)>assumedzero) for the last branch we find it is equivalent to abs(x)>assumedzero and we discover that the last branch maintains (state = iterating). This results in the last branch being split into two branches:

```
  [] ¬zerodivattempted∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd)>tolerance∧
     itcount-1≠maxits∧abs(x)>assumedzero → itcount := itcount+1; oldx := x; x := x-fx/fd
  [] ¬zerodivattempted∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd)>tolerance∧
     itcount-1≠maxits∧abs(x) ⩽ assumedzero → itcount := itcount+1; oldx := x; x := x−fx/fd
```

We may therefore directly apply the DRG transformation to obtain:

```
  ......
  itcount := 0; state := iterating;
  if(state = iterating)∧(abs(oldx)>assumedzero) →
     Evaluatef(x, fx, zerodivattempted);
     Evaluatefdashed(x, fd, zerodivdashedattempted);
     do zerodivattempted∧¬zerodivdashedattempted∧abs(fd)>assumdzero∧
        abs(x*fx/fd)>tolerance∧itcount-1≠maxits∧abs(x)>assumedzero →
           itcount := itcount+1; oldx := x; x := x-fx/fd;
           Evaluatef(x, fx, zerodivattempted);
           Evaluatefdashed(x, fd, zerodivdashedattempted)
     od;
     if zerodivattempted → state := zerodivinf
     [] ¬zerodivattempted∧zerodivdashedattempted → state := zerodivinfdashed
     [] ¬zerodivattempted∧¬zerodivdashedattempted∧abs(fd) ⩽ assumdzero →
              state := flatspotmet
     [] ¬zerodivattempted∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd) ⩽ tolerance →
              itcount := itcount+1; oldx := x; x := x-fx/fd; state := converged
     [] ¬zerodivattempted∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd)>tolerance∧
        itcount-1 = maxits → itcount := itcount+1; oldx := x; x := x-fx/fd; state := maxitsreached
     [] ¬zerodivattempted∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd)>tolerance∧
        itcount-1=maxits → itcount := itcount+1; oldx := x; x := x-fx/fd
     fi
  fi;
  if state = iterating → state := xtoonearzero fi;
  case
     state = converged: ...; /* converged */
     state = maxitsreached: ...; /* maxitsreached */
     state = flatspotmet: ...; /* flatspotmet */
     state = zerodivinf: ...; /* zerodivinf */
     state = zerodivinfdashed: ...; /* zerodivinfdashed */
     state = xtoonearzero: ...; /* xtoonearzero */
end
```

This restructuring yields not only a rationalized loop but it also indicates a number of quality defects in the original loop:

- The flag *state* can be completely removed because it is used only to identify the exit status (see the *CASE* statement).
- The last *IF* statement *IF* state = iterating *THEN* state := xtoonearzero can be reached only from the last branch of the MBS because it is guarded by *state = iterating* and the last branch has a postcondition *state = iterating*. We may therefore use the branching transformation to combine them;
- The variable *oldx* is totally redundant and can therefore be removed because it is never used.

We then finally end up with the structure:

```
  ......
  itcount := 0;
  IF abs(...)>assumedzero THEN                         /* the initial value of oldx */
  BEGIN
    Evaluatef(x, fx, zerodivattempted);
    Evaluatefdashed(x, fd, zerodivdashedattempted);
    WHILE ¬zerodivattempted∧¬zerodivdashedattempted∧abs(fd)>assumdzero∧
        abs(x*fx/fd)>tolerance∧itcount-1≠maxits∧abs(x)>assumedzero DO
      BEGIN
        itcount := itcount+1; x := x-fx/fd;
        Evaluatef(x, fx, zerodivattempted);
        Evaluatefdashed(x, fd, zerodivdashedattempted)
      END;
    CASE
    zerodivattempted: ...                                    /* zerodivinf */
    ¬zerodivattempted∧zerodivdashedattempted: ...            /* zerodivinfdashed */
    ¬zerodivattempted∧¬zero...∧abs(fd)˙assumdzero: ...       /* flatspotmet */
    ¬zerodivattempted∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd) ⩽ tolerance:
          itcount := itcount+1; oldx := x; x := x-fx/fd; .../* converged */
    ¬zero...∧¬zero...∧abs(fd)>assumdzero∧abs(x*fx/fd) ⩽ tolerance∧itcount-1 = maxits:
          itcount := itcount+1; oldx := x; x := x-fx/fd; .../* maxitsreached */
    ¬zero...t∧zero...∧abs(fd)>assumdzero∧abs(x*fx/fd) ⩽ tolerance∧itcount-1≠maxits:
          itcount := itcount+1; oldx := x; x := x-fx/fd; .../* xtoonearzero*/
    END
END
```

As with the previous example this loop is implemented in the most straightforward way using an Ada loop-statement which has a single exit in the middle. In this instance the textually repeated structure involves the two procedure calls Evaluatef and Evaluatefdashed.

7. CONCLUSION

We have introduced a loop re-engineering process, called *loop rationalization*, that may be used to transform loop structures that contain multiple-exits and flags. The advantage of this approach, over other alternatives that have been proposed, is that it is securely based on a formal model involving strongest postcondition calculations. The use of strongest postconditions allows us to derive the full benefits of directly using the semantics of the program structure that is being transformed—such benefits are not possible using simple transformation-based restructuring methods. The only other recent formal work on exits has been published by King and Morgan [26]. The focus of that work is upon formally accommodating exits and exceptions. These objectives are quite different from what we have tried to do here.

The processes we have developed can be used to detect and remove a number of quality, efficiency and reliability defects from program structures. While the focus here has been on dealing with loops containing multiple exits the method is also able to suggest substantive optimizations for branched loop structures that do not contain multiple exits. The proposed processes represent a set of general, language-independent, and widely applicable techniques for improving the quality of loops and programs. Loop rationalization may also serve as a preprocessing step for the more general loop normalization process [20] which is needed to remove dynamic logical redundancy from program structures. Loop rationalization, in concert with loop normalization, provides a powerful approach to enhancing the quality of loop structures. Both processes have the potential for automated implementation. What is more, loop rationalization is quite straightforward to apply manually. It can render difficult code systematically manageable without the usual tedium of pouring over existing complex program structures. The strategy involves first taming such structures using the methods we have suggested and only then proceeding to analyse the code for its intent.

REFERENCES

[1] Arblaster, A. T., Sime, M. E. and Green, T. R. (1979) Jumping to Some Purpose. *Comp. J.*, **22**, 105–109.
[2] Atkinson, L. V. (1978) Know the State Your're in. *Pascal News*, **13**, 66–69.
[3] Atkinson, L. V. (1979) Pascal Scalars are State Indicators. *Softw. Pract. and Exp.*, **9**, 427–431.
[4] Atkinson, L. V. (1984) Jumping about and Getting into a State. *Comp. J.*, **27**, 42–46.
[5] Cristian, F. (1980) *Exception Handing and Software-Fault Tolerance*. 10th Annual International Symposium on Fault-Tolerant Computing, Kyoto, Japan, pp. 97–103.
[6] Dromey, R. G. (1995) A Model for Software Product Quality. *IEEE Trans. Soft. Eng.*, **21**(2), 143–152.
[7] Hill, I. D. (1980) Jumping to Some Purpose. *Comp. J.*, **23**, 94.
[8] Goodenough, J. B (1975) Exception Handling: Issues and a Proposed Notation. *CACM*, **18**, 683–696.
[9] Inglis, J. (1982) Jumping to Some Purpose. *Comp. J.*, **25**, 495.
[10] Lee, P. A. (1983) Exception Handling in C Programs. *Softw. Pract. Exp.*, **13**, 389–405.

[11] Levin, B. H. (1977) *Program Structure for Exceptional Condition Handling*. Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, USA.

[12] Liskov, B. H. and Snyder,A. (1979) Exception Handing in CLU. *IEEE Trans. Softw. Eng*., **SE-56**, 546–558.

[13] Luckham, D. C. and Polak, W. (1980) Ada Exception Handing: an Axiomatic Approach. *ACM/TPLS.*, **2**, 225–233.

[14] MacLaren, M. D. (1977) Exception Handing in PL/I. *SIGPLAN NOTICES*, **12**, 101–104.

[15] Missala, M. and Rudnicki, P. (1982) Jumping to Some Purpose. *Comp. J.*, **25**, 286.

[16] Pan, S. and Dromey, R. G. (1996) *A Formal Basis for Eliminating GOTO Statements*. *Comp. J*., **39**, 203–214.

[17] Pan, S. and Dromey, R. G. (1996) Beyond Structured Programming. International Conference on Software Engineering, ICSE-18, Berlin, March.

[18] Robinson, G. L. (1980) Jumping to Some Purpose. *Comp. J*., **23**, 288.

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

[20] Pan, S. and Dromey, R. G. (1993) *Loop Normalization*. Report, Griffith University, Australia.

[21] Pan, S. (1995) *Software Quality Improvement, Specification Derivation and Quality Measurement Using Formal Methods*. Ph.D. Thesis, Griffith University, Australia.

[22] Dromey, R. G. and McGettrick, A. D. (1992) On Specifying Software Quality. *Soft. Qual. J*., **1**(1), 45–74.

[23] Pan, S. and Dromey, R. G. (1994) *A Formal Basis for Measuring Software Quality Using Formal Methods*. 17th Annual Comp. Sci. Conf., Christchurch, New Zealand.

[24] McCabe, T (1976) A Complexity Measure. *IEEE Trans. Softw. Eng*., **SE-1**, 312–327.

[25] Robinson, G. L. (1983) Jumping to Some Purpose. *Comp. J*., **26**, 95.

[26] King, S. and Morgan, C. (1995) Exits in the Refinement Calculus. *Formal Aspects of Computing,*. **7**, 54–76.