BEYOND STRUCTURED PROGRAMMING

S. Pan, R.G. Dromey,
Software Quality Institute,
Griffith University,
Nathan, Qld,
AUSTRALIA 4111

Abstract

Structured programming principles are not strong enough to control complexity and guarantee high reliability of software at the module level. Stronger organizing principles and stronger properties of components are needed to make significant gains in the quality of software. Practical proposals, based on the definition of normal forms which have a mathematical/logical foundation, are suggested as a vehicle for constructing software that is both simpler and of higher quality with regard to clearly defined and justifiable criteria.

1. Introduction

Stronger principles of programming at the module level are needed if we are to make significant gains in controlling the complexity of software, improving its reliability and its overall quality. Structured programming [3], modularization techniques and information hiding [15] and abstract data types [9] have all made important contributions to this end. However when we ask things like:

- What belongs and what does not belong in a loop body,
- Is there an implementation for this algorithm that has a smaller McCabe Number
- What is the ideal structure for a sequence of statements,
- How many local variables are needed to implement this algorithm, or
- Is a given module cohesive or should it be split into two or more smaller modules.

we find there are few systematic guiding principles that help us to satisfactorily respond to such questions. Clearly it is desirable to have sound and reasoned answers to questions of this sort. It is necessary to turn to other related disciplines to find the clues that will help us respond to such questions.

In abstract algebra[1], logic[8], database theory[16] and other fields, *normal forms* are widely used to settle issues of structure, to define ideals of form and to provide efficiencies and economies of expression. In contrast, in computing, there has been little interest in employing normal forms. The tremendous advantages that have been gained from using normal forms in related fields suggests that it might be well worth our while to try to develop a normal form theory for program structures. Existing structuring proposals have not done enough to control complexity and improve the reliability and overall quality of software.

The fact that program structures can be modelled using graph theory and that the semantics of program statements may be modelled as predicate transformers [2,4,13] provides an excellent opportunity to search for normal forms that may capture ideals of form and composition that can point the way to the construction of software that is simpler, more reliable, more efficient and easier to maintain. We will also show that such techniques offer an excellent opportunity to put measurement of the relative quality of software and its complexity on a much sounder mathematical footing.

In putting forward proposals for the use of normal forms in programming we are not interested in projecting the theory of program structure into an obscure realm of abstract mathematics. What we are interested in doing is providing sound, mathematically grounded, practical principles that can be routinely used by software engineers to guide the construction, and measurement of key properties of software that impact its quality. These principles define ideals of form according to clearly enunciated criteria and therefore also open up the possibility of re-engineering poor quality software to a normal form. What we will put forward here is one set of proposals for normal forms. There may certainly be other alternatives. It is our hope that the present work will encourage others to look at the structure of software from this perspective.

2. Normal Forms and Program Structure

In applying the concept of normal forms to program structure there are three possible avenues of exploration that we can pursue. We can start by looking at possible normal forms for *simple and composite statements* and statement components. This avenue also takes into account a number of issues of composition. Another fruitful place to look is at *modules*. There have been proposals by Myers and others, relating to coupling and cohesion [12]. These proposals, like the present work, could be interpreted as suggestions for normal forms. It is however possible to extend these proposals by taking a more rigorous look at module cohesion and the use of *variables* within modules.

The application of normal forms corresponds to the imposition of more stringent logical and structural constraints on software components. In one sense, we are suggesting that more order should be imposed on program structures. Our intent here is to rigorously define a set of normal forms for program structures. Before presenting our suggestions let us first look at the most well-known form associated with program structure - the proposals for structured programming which constitute a normal form. They require that every type of executable statement has a *single* point of entry and exit. Imposition of this constraint has made an important contribution to simplifying the structure of imperative programs. Languages like Modula-2 have chosen to use a language architecture that directly implements and enforces structured programming.

An obvious question to ask is: *is structured programming the only normal form relevant and applicable to program structure?* We contend that this is not the case. That, in fact, there are a number of compatible normal forms relevant to program structure which can be superimposed to yield further important refinements for improving the structural quality of programs. The suggestions we will make are intended to simplify program structure and at the same time make the software more reliable.

A significant contribution to program structure is made by the statements that change the flow of control: the iterative structures and the selection statements. It is the nested composition of such structures which significantly adds to the complexity of programs. What we are interested in is a normal form that provides strict guidelines for controlling the complexity of such structures. Single point of entry and exit is not strong enough to prevent such structures from becoming rather complex. At first sight it would seem that there is not much we can do to reduce the complexity of such structures. Proposals to date have concentrated only on normal forms that exploit syntax and, as such, are not strong enough to control complexity. The key lies in proposing normal forms that exploit *semantics*. Semantics allows us to impose more stringent constraints on structures, to define forms that are ideal with respect to some clearly specified criterion and identify semantics-preserving transformations to convert delinquent structures to normal forms. The resulting normal form, in each case, is not usually a property of a statement per se, but of a statement in a specific semantic context.

In seeking to define normal forms we will use *constructive* definitions. Using this approach to assess whether a given structure is in its normal form it is necessary to carry out a two-step process:

- first we transform the [branched] statement (simple or compound) to its base form, and then
- we perform a semantic analysis to determine whether the structure is already normalized

3. Normal Forms for Statements

It is possible to define normal forms for each type of statement, both simple and compound that we encounter in conventional imperative programming. These normal forms express an *ideal* for representing the given statement with respect to well-defined and testable criteria. They are designed to achieve four outcomes:

- ensure the inherent consistency and ***problem-independent correctness*** of the statement is preserved,
- minimize its complexity,
- maximize its efficiency, and
- maximize analyzability.

An additional requirement is that it should be possible to automate the process of transforming a statement into its normal form. We will now examine the normal forms for several key program components to illustrate the method.

3.1 Assignments

Assignments are the simplest building blocks of imperative programs. An obvious syntactic requirement for an assignment is that all variables used in its expression should be initialized and there should be no side-effects in the expression. Apart from these requirements there is only one semantic obligation on an assignment. It is that assignments should always advance a computation. By this we require that the postcondition established by an assignment should not be equivalent to its precondition. This constitutes a normalization condition for assignments. If an assignment does not satisfy this requirement it is redundant and may be removed from the computation without consequence. We may express this normalization condition formally using strongest postconditions. We define sp(P,S) to represent the strongest postcondition [2,13] established by the statement S when it executes in the context of the precondition P. For an assignment S executing in the context of the precondition P we require

$$\boxed{\textbf{NAS-1:}\ \neg(sp(P,S) \equiv P)}$$

to be satisfied for the assignment to be normalized. Otherwise such an assignment S must be a self-assignment, e.g. x:=x or establish an already-established condition. It is straightforward to do a strongest postcondition calculation to determine whether an assignment is normalized.

3.2 If-Statements

It is important to apply semantic normalization techniques to branch statements to avoid all the potential insecurities and correctness problems with such statements. To ensure that a branch structure is normalized three types of condition must be considered: the precondition P, the set of branch guards $\{ C_1, C_2, \ldots, C_N \}$ and the set of conditions $\{ D_1, D_2, \ldots, D_N \}$ that apply after each branch has executed. Here we consider branch statements of the form [4]:

```
        $\{P\}$
        $\textit{if}\ C_1 \rightarrow S_1$        $\{D_1\}$
        $[]\ C_2 \rightarrow S_2$        $\{D_2\}$
        $[]\ \cdots \rightarrow \ldots$
        $[]\ C_N \rightarrow S_N$        $\{D_N\}$
        $\textit{fi}$
```

For such a statement to be normalized we suggest that the following conditions must be satisfied in concert.

Normalized Multiple-Branch Statement:

NMB-1.	$P \Rightarrow C_1 \vee C_2 \vee \ldots \vee C_N$
NMB-2.	$\forall i \forall j\ ((i,j \in [1,N] \wedge j \neq i) \Rightarrow (C_i \Rightarrow \neg C_j))$
NMB-3.	$\forall i\ (i \in [1,N] \neg\ (P \equiv D_i))$
NMB-4.	A variable unassigned in P that is assigned in *at least one* branch must be assigned in *all* branches.
NMB-5.	$\neg\exists i\ ((i \in [1,N])\ P \Rightarrow C_i)$ $\wedge \neg\exists i\ ((i \in [1,N])\ \ P \Rightarrow \neg C_i)$
NMB-6.	$\neg\exists i\ (sp(P \wedge C_1, S_1) \vee \ldots \vee sp(P \wedge C_N, S_N))$ $\equiv sp(P, S_i))$
NMB-7.	$sp(P \wedge C_1, S_1) \vee \ldots \vee sp(P \wedge C_N, S_N)$ is conjunctive

NMB-1
The first normalization condition ensures that the if-statement is able to accommodate all possible initial states under which it is required to execute. That is, there will be no initial state defined by P for which all branch guards are false.

NMB-2
For reasons of efficiency and consistency it is desirable that ,for any given precondition, the guard for only one branch should be satisfied. What this means in formal terms is that for all branch guards, the satisfaction of any guard means that all other branch guards are not satisfied. The second normalization condition NMB-2 captures this requirement

NMB-3
A non-empty branch statement $C_i \rightarrow S_i\ \{ D_i \}$ in an if-statement is redundant when that statement establishes a postcondition D_i that is equivalent to the precondition P for the if-statement. A normalized if-statement should contain no redundant branches.

NMB-4
A significant insecurity in if-statements in most languages is that they do not enforce the requirement that a variable which is unassigned in the precondition P, and is assigned in at least one branch of the if-statement it must be assigned in every branch of the if-statement. If this requirement is not fulfilled it leaves open the possibility that on termination of the if-statement in some circumstances the variable will be undefined while in other cases it will be defined. This represents a correctness problem with the if statement. The normalization requirement is designed to avoid this problem.

NMB-5
While the first normalization condition NMB-1 guarantees that at least one branch is reachable under the deterministic precondition P it does not however guarantee that all branches are reachable under P. The normalization condition NMB-5 requires that all branches are reachable under the precondition P.

NMB-6
To be normalized an overall branch structure must not establish a strongest postcondition that is equivalent to the strongest postcondition established by one, or a strict subset, of its branches or by a simpler statement S that involves no branches.

To illustrate NMB-6, let us consider Manna's Abstract Program **AP** [10] under the precondition **true**:

```
    $\textit{if}\ p(y) \rightarrow x{:=}y;\ \textbf{\textit{do}}\ \neg p(x) \rightarrow x{:=}f(x)\ \textbf{\textit{od}}$
    $[]\ \neg p(y) \rightarrow x{:=}a;$
        $\textit{if}\ p(x) \rightarrow$ /*$\{\neg p(y) \wedge x{=}a \wedge p(x)\}$*/
                    $\textbf{\textit{do}}\ \neg p(x) \rightarrow x{:=}f(x)\ \textbf{\textit{od}}$
        $[]\ \neg p(x) \rightarrow x{:=}f(x);$  /*$\{\neg p(y) \wedge x{=}f(a) \wedge \neg p(a)\}$*/
            $\textit{if}\ p(x) \rightarrow x{:=}a;$ /*$\{\neg p(y) \wedge x{=}a \wedge \neg p(a) \wedge p(f(a))\}$*/
                        $\textbf{\textit{do}}\ \neg p(x) \rightarrow x{:=}f(x)\ \textbf{\textit{od}}$
            $\textbf{\textit{fi}}$
        $\textbf{\textit{fi}}$
    $\textbf{\textit{fi}}$
```

The last branch statement *if* $p(x) \rightarrow x{:=}a$; ***do*** $\neg p(x) \rightarrow x{:=}f(x)$ ***od fi***, after execution of its internal statement under the precondition $\neg p(y) \wedge x{=}f(a) \wedge \neg p(a)$, yields $\neg p(y) \wedge x{=}f(a) \wedge \neg p(a) \wedge \neg p(f(a))$ for skip and $\neg p(y) \wedge x{=}a \wedge \neg p(a) \wedge p(f(a))$ for the execution of the loop ***do*** $\neg p(x) \rightarrow x{:=}f(x)$ ***od*** because after the first iteration the loop guard $\neg p(x)$ fails. Therefore:

$$\begin{aligned}
&sp(\neg p(y) \wedge x{=}f(a) \wedge \neg p(a), \\
&\quad \textit{if}\ p(x) \rightarrow x{:=}a;\ \textbf{\textit{do}}\ \neg p(x) \rightarrow x{:=}f(x)\ \textbf{\textit{od fi}}) \\
&\equiv (\neg p(y) \wedge x{=}f(a) \wedge \neg p(a) \wedge \neg p(f(a))) \\
&\quad \vee (\neg p(y) \wedge x{=}a \wedge \neg p(a) \wedge p(f(a))) \\
&\equiv \neg p(y) \wedge x{=}f(a) \wedge \neg p(a) \\
&\equiv sp(\neg p(y) \wedge x{=}f(a) \wedge \neg p(a),\ \emptyset)
\end{aligned}$$

According to the condition NMB-6 the branch statement can be replaced by the empty statement, $\emptyset$. More generally a normalized branch statement should satisfy the requirement that there exists no simpler statement S than the branch statement such that:

$$sp(P\wedge C_1, S_1) \vee sp(P\wedge C_2, S_2) \vee \ldots \vee sp(P\wedge C_N, S_N)) \equiv sp(P, S)$$

NMB-7

The normalization conditions NMB1-6 define a number of requirements for a normalized branch statement. The normalization condition NMB-7 differs from these requirements in that it requires that a branch statement should have a unified functionality. This condition may be expressed by requiring that the postcondition should be equivalent to a conjunctive condition if the precondition is conjunctive.

To understand this, consider a two-branch statement { $a(x)\wedge b(z)$ } ***if*** $x=0 \rightarrow y:=M$ [] $x\neq 0 \rightarrow y:=M-1$ ***fi*** as an example. Obviously, the postcondition $a(x)\wedge b(z) \wedge ((x=0\wedge y=M) \vee (x\neq 0\wedge y=M-1))$ cannot be simplified to a conjunctive form without any disjunctive term. Often programmers use this sort of device to set another variable y that can be used in another branch test in the statements that follow. However we may use $x=0$ and $x\neq 0$ to replace this additional test. This is possible because in one sense this branch statement establishes another flag, y, from the original flag x.

When the disjunctive postcondition $D_1 \vee D_2$ can be simplified into an equivalent conjunctive form D, there are two cases that apply. One is that the weakest branch postcondition D_i is equivalent to D, and another D_j implies D (as $D_1 \vee D_2 \equiv D \vee D_j \equiv D$, where $j\in[1,2]$). Another case is that both of the branch postconditions D_1 and D_2 imply D, and each of them has an extra part R_i, i.e., $D_i \equiv D \wedge R_i$, for $i\in[1,2]$, which satisfies $R_1 \vee R_2 \equiv$ **true**. We suggest that the normalized branch statement should be classified according to the first case, rather than the second case. Because, from a theoretical viewpoint, these R_i do not appear in the postcondition, they may be used only for branching purposes (otherwise they are redundant). From a technical view point, if these R_i are needed for the branch statement as branch conditions, the branch conditions or their modifications of the original branch statement that produces these R_i may replace them. For example, ***if*** $x=0 \rightarrow x:=1$; flag:=**true** [] $x\neq 0 \rightarrow x:=x+1$; flag:=**false** ***fi*** is not a normalized branch statement under any precondition P(x) because

sp(P(x), ***if*** $x=0 \rightarrow x:=1$; flag:=**true**
[] $x\neq 0 \rightarrow x:=x+1$; flag:=**false** ***fi***)
$\equiv P(x-1)\wedge(x=1\wedge flag \vee x-1\neq 0\wedge\neg flag)$
$\equiv P(x-1)\wedge((x=1)\leftrightarrow flag)\wedge(x=1\vee x\neq 1)$
$\equiv P(x-1)\wedge((x=1)\leftrightarrow flag)$

From a logical viewpoint, the flag is equivalent to the predicate x=1, which means there is redundancy present. From a programming viewpoint, when the flag is needed in the preceding program, the predicate x=1 may be used to replace it. In contrast,

if $A[i+1]\leq A[p] \rightarrow i:=i+1$ [] $A[i+1]>A[p] \rightarrow i,p:=i+1,i+1$ ***fi***

is a normalized branch statement under the precondition $\exists p\in[1,i]\forall t\in[1,p-1]\forall s\in[p+1,i]\ A[t]<A[p]\wedge A[s]\leq A[p]$ because

sp($\exists p\in[1,i]\forall t\in[1,p-1]\forall s\in[p+1,i]A[t]<A[p]\wedge A[s]\leq A[p]$,
if $A[i+1]\leq A[p] \rightarrow i:=i+1$ [] $A[i+1]>A[p] \rightarrow i,p:=i+1,i+1$ ***fi***)
$\equiv (\exists p\in[1,i]\forall t\in[1,p-1]\forall s\in[p+1,i]A[t]<A[p]\wedge A[s]\leq A[p]) \vee (\forall t\in[1,i-1]A[t]<A[i]\wedge p=i)$
$\equiv \exists p\in[1,i]\forall t\in[1,p-1]\forall s\in[p+1,i]A[t]<A[p]\wedge A[s]\leq A[p]$ (as the latter implies the former)

3.3 Loops

The use of semantics allows us to clearly and precisely define a number of important structural and correctness requirements for loops. We will divide our discussion of the normalization requirements into two parts, the first part consists of requirements that apply to all loops and the second consists of those requirements that apply to loops that contain branched loop bodies, i.e., they contain IF-statements and/or loops). In our discussion we will consider a general loop ***do*** G $\rightarrow$ S ***od*** that executes under a precondition P. These requirements extend *beyond* the usual requirements for loop correctness [4].

3.3.1 Basic Normalization Conditions for Loops: The general requirements are designed to ensure that the context defined by the precondition P and the guard G, really warrants the use of a loop that has the potential to execute more than once and to terminate. The precondition P here is not the loop invariant but rather a condition (established by the loop initialization) which will always *strictly imply* the strongest loop invariant for the loop.

Basic Normalization Conditions for Loops:
LBN-1. $\neg(P \Rightarrow \neg G)$
LBN-2. $\neg$(sp(P, ***do*** G $\rightarrow$ S ***od***) $\equiv P \wedge \neg G$)
LBN-3. $\neg(sp(P \wedge G, S) \Rightarrow \neg G)$
LBN-4. $\neg((sp(I \wedge G, S) \Rightarrow G)$

LBN-1

The first normalization condition LBN-1 requires that the loop is reachable for all possible initial states under which it must execute. That is, there must be no initial state defined by a non-grounded precondition P for which the loop guard is false.

LBN-2

The condition LBN-2 deals with the efficiency of a loop. It is different to LBN-1 in that the loop may execute a number of times, however it demands that the loop establishes more than simply the negation of the loop guard without changing the precondition P. For example, the loop { P } ***do*** $i\neq N \rightarrow i:=i+1$ ***od*** should be avoided because it produces only the condition i=N even though it may execute a number of times.

LBN-3
Any loop, to be deserving of that status should have the potential to execute more than once for a given non-grounded precondition P and guard G. This requires not only that the loop is reachable, but also that it is executable at least twice. What this means in formal terms is that the postcondition from the first iteration should not imply the negation of the loop guard.

LBN-4
Every loop executing under its strongest loop Invariant I should have a structure which guarantees, that when the loop body executes it does not establish a condition which implies the loop guard G. Obviously, if it does, the loop is non-terminating.

3.3.2 Loops with Branched Bodies:

Loops structures that contain branched bodies are often regarded as the most complex, the most difficult to analyze, and the most likely to contain logical flaws. It is therefore important to see how the concept of normalization may be employed to make such structures more tractable.

Any loop with a branched body can be transformed into the following equivalent form (in many instances S will be empty)[14]:

```
{P}
do G →
    S;
    if C1 → S1
    [] C2 → S2
    [] ... → ...
    [] CN → SN
    fi
od
```

The basic normalization conditions for loops apply for this loop structure. Additional normalization constraints that apply to branch structures are as follows:

Branched Loop Normalization

Upon loop entry: each branch should be reachable, i.e.,
LBB-1a. $\neg \exists i\ ((i \in [1,N])\ sp(P \wedge G, S) \Rightarrow C_i)$
LBB-1b. $\neg \exists i\ ((i \in [1,N])\ sp(P \wedge G, S) \Rightarrow \neg C_i)$

After each iteration: each branch should be reachable, i.e.,
LBB-2a. $\neg \exists i\ \forall j((i,j \in [1,N])\ sp(sp(G, S) \wedge C_j, S_j) \Rightarrow C_i)$
LBB-2b. $\neg \exists i\ \forall j((i,j \in [1,N])\ sp(sp(G, S) \wedge C_j, S_j) \Rightarrow \neg C_i)$
Loop Progress: each branch should decrease its variant function

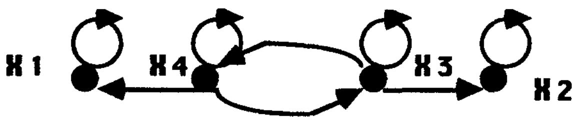
If $sp(P \wedge G, S)$ or $sp(sp(G, S) \wedge C_j, S_j)$ implies one of the branch conditions, then other branch conditions cannot be reached. Also if either of these conditions implies the negation of a branch condition, then that branch is unreachable. A normalized, branched loop body avoids all of these problems. What LBB-2a,2b tell us is that each branched loop should have a strongly connected branch successor graph. For example, the following List Merge [11]

```
out:=Nil;
do ($in_1$≠Nil)∨($in_2$≠Nil) →
  if ($in_2$=Nil)∨(($in_1$≠Nil)∧($in_2$≠Nil)∧($in_1$.head≤$in_2$.head))
      → out:=out++$in_1$.head; $in_1$=$in_1$.tail
  [] ($in_2$≠Nil)∧(($in_1$=Nil)∨($in_2$=Nil)∨($in_1$.head≥$in_2$.head))
      → out:=out++$in_2$.head; $in_2$=$in_2$.tail
  fi
od
```

contains four branches within its loop body:

```
if $in_2$=Nil → out:=out++$in_1$.head; $in_1$:=$in_1$.tail
[] ($in_2$≠Nil)∧($in_1$=Nil) → out:=out++$in_2$.head; $in_2$:=$in_2$.tail
[] ($in_2$≠Nil)∧($in_1$≠Nil)∧($in_1$.head≤$in_2$.head) →
                    out:=out++$in_1$.head; $in_1$:=$in_1$.tail
[] ($in_2$≠Nil)∧($in_1$≠Nil)∧($in_1$.head>$in_2$.head)) →
                    out:=out++$in_2$.head; $in_2$:=$in_2$.tail
fi
```

It satisfies the conditions LBB-1a and LBB-1b, however it fails to match LBB-2a and LBB-2b because after execution of the first or second branch, the control-flow will remain in the same branch or terminate. Also after execution of the third branch, the control-flow will only enter either the second or the third or the fourth branch, or terminate; and so on. We may use the following branch successor graph (BSG) to illustrate this:



These quality defects are caused by poor decomposition of the problem. We may remove these defects by formally transforming this algorithm into the following equivalent form:

```
out:=Nil;
do ($in_1$≠Nil)∧($in_2$≠Nil) →
      if $in_1$.head≤$in_2$.head → out:=out++$in_1$.head;
                                  $in_1$=$in_1$.tail
      [] $in_1$.head≥$in_2$.head → out:=out++$in_2$.head;
                                  $in_2$=$in_2$.tail
      fi
od;
do ($in_1$≠Nil) → out:=out++$in_1$.head; $in_1$=$in_1$.tail od;
do ($in_2$≠Nil) → out:=out++$in_2$.head; $in_2$=$in_2$.tail od
```

The transformations required to achieve this result correspond to splitting the original BSG into its three strongly connected components (corresponding to the three loops). A more detailed treatment of loop reengineering is given elsewhere[14]

3.4 Non-Iterative Sequences

Simply composing a compound statement of normalized statements is not strong enough to guarantee that the compound statement is in a normal form. Here we consider non-iterative sequences that are composed of assignment statements, input/output statements and if-statements. There is a potential for un-normalized sequences to contain redundant assignments, redundant tests and involve the use of variables for more than a single purpose. All of these defects may be removed by a sequence of transformations that include several optimizations. We will now examine the optimized structures of non-iterative statement sequences, which contain no logical redundancy. An EBNF definition will be given for each of these structures.

We start with the structurally simplest sequence. Consider any non-iterative statement sequence consisting of assignments and ***write***-statements only. We can apply simple transformations [13] to remove all redundant assignments from the sequence. The resulting sequences have the property that no variable is assigned twice. Hence we have:

<u>NNIS-1</u> *Minimum Optimization Structure* $<O_{mos}>$:
This is a redundancy-free sequence containing assignments and ***write***-statements, where any variable is assigned only once. In EBNF we have:

$<O_{mos}>$ ::= ∅ | x:=E {; $<O_{mos}>$} | ***write***(E) {; $<O_{mos}>$}

The order of the statements in a $<O_{mos}>$ sequence can be freely interchanged using a straightforward set of substitution rules. For example, the following two three-statement sequences establish exactly the same postcondition:

x:=a+b; y:=u*v; s:=x-y ⊨ s:=a+b-u*v; x:=a+b; y:=u*v

We may extend the complexity of $<O_{mos}>$ sequences to include any non-iterative statement sequence consisting of assignments and I/O statements only. When all ***read***-statements involve different ***read***-variables, there exist transformations that allow all ***read***-statements to be relocated at the front of the sequence. When there exist two ***read***-statements involving the same ***read***-variable, a reassignment rule is used to rename the first ***read***-variable and its usages by a fresh variable. This allows all ***read***-statements to be placed at the front of the sequence. The rest of the sequence contains only assignments and ***write***-statements that form a $<O_{mos}>$ sequence. This leads to a new optimized sequence that satisfies the following definition:

<u>NNIS-2</u> *Bounded Minimum Optimization Structure* $<O_{bmos}>$:
This is a redundancy-free sequence, which starts with a number of (one or more) ***read***-statements followed by a sequence in $<O_{mos}>$, where each variable is assigned once either by a ***read***-statement or an assignment. In EBNF we have:

$<O_{bmos}>$ ::= ***read***(x); $<O_{mos}>$ | ***read***(x) {; $<O_{bmos}>$},

We may extend any non-iterative statement sequence consisting of assignments, I/O statements to include if-statements. When all branch-guards of an if-statement depend on at least one ***read***-variable, these branch-guards are not pre-determined for arbitrary input. Several transformations [13] exist which enable us to locate such an if-statement in front of all statements that follow the ***read***-statements which initialize at least one ***read***-variable that appears in the branch-guards. The encapsulating structure starts with a number of (one or more) ***read***-statements followed by an if-structure, called the *Bounded Branch Structure*. In this branched structure each internal statement (branch body) is either another bounded branch structure or a sequence in $<O_{mos}>$ or $<O_{bmos}>$ or $<O_{bbos}>$.

<u>NNIS-3</u> *Bounded Branched Optimization Structure* $<O_{bbos}>$:
This is a redundancy-free, recursively defined sequence that starts with a number of (one or more) ***read***-statements followed by a branch structure, in which each branch-guard involves at least one ***read***-variable assigned by the previous ***read***-statements and each internal statement is a sequence in $<O_{mos}>$ or $<O_{bmos}>$ or $<O_{bbos}>$.

$<O_{bbos}>$ ::= ***read***(x); ***if*** C_1(x,...) → <Body> [] ... [] C_m(x,...) → <Body> ***fi***
| ***read***(x) {; $<O_{bbos}>$},
where <Body> ::= $<O_{mos}>$ | $<O_{bmos}>$ | $<O_{bbos}>$ and where any variable in any $<O_{bbos}>$ structure is assigned only once for each execution path, and C_i(x,...) for $i \in [1,m]$ must involve all pre-fixed ***read***-variables;

When there exists an if-statement whose branch-guards are independent of any ***read***-variable in the sequence, transformations exist which enable us to locate this branch statement at the front of the sequence. This constructs a branch structure directly. In such a branch structure, each internal statement can be simplified into a sequence in $<O_{bbos}>$ or $<O_{bmos}>$ or $<O_{mos}>$.

<u>NNIS-4</u> *Branch Optimization Structure* $<O_{bos}>$:
This is a redundancy-free branch structure such that all its internal branch statements are sequences in either $<O_{bbos}>$ or $<O_{bmos}>$ or $<O_{mos}>$.

$<O_{bos}>$ ::= ***if*** C_1 → <Body> [] ... [] C_m → <Body> ***fi***, where any variable in any $<O_{bos}>$ structure is assigned only once for each execution path;

Sequences in $<O_{bos}>$ and $<O_{bbos}>$ have similar properties to sequences in $<O_{mos}>$ and $<O_{bmos}>$. Each variable is assigned once (for each execution path) either by a ***read***-statement or an assignment and the order of any ***read***-statements that bounds a branch structure can be freely changed by appropriate variable substitution within their range.

Strongest postcondition calculations, together with transformations in line, provide a means to restructure any non-iterative statement sequence into an equivalent optimized, redundancy-free sequence in $<O_{mos}>\cup<O_{bmos}>\cup<O_{bbos}>\cup<O_{bos}>$. These four structures define the *Optimized Structures* for any non-iterative statement sequence.

4. Variable Normalization

The way variables are used can have a significant impact on the quality of software. Single assignment languages like SISAL overcome most of these problems. Here we will consider requirements that may be applied to imperative programs that do not exploit single assignment. The consistency principle tells us that a variable should only be used for a single purpose. Local variables are frequently used to help achieve this ideal. This does not mean, however, that local variables should be declared freely. For example, with the module swap(x,y), the ideal segment should use one local variable, i.e., t:=x; x:=y; y:=t, rather than two local variables, such as t1:=x; t2:=y; x:=t2; y:=t1. Some formality is needed to resolve this sort of problem.

Local variables are used to record state information at critical points in a computation. It is not easy however to determine the number of local variables for the ideal representation of a given algorithm or program. Instead, we may use calculations to define the maximum number of local variables to model a particular set of state-recording requirements for any algorithm. When a program uses more local variables than the maximum required number, then its local variable set should be reduced by normalization.

Before defining the maximum number of local variables needed, we need to convert any assignment sequence by substitution into an equivalent multiple assignment $\boldsymbol{x}:=\boldsymbol{f}$. For example, with two output variables x and y, the sequence t1:=x; t2:=y; x:=t2; y:=t1 produces (t1, t2, x, y):=(x, y, y, x). Since $\boldsymbol{x}:=\boldsymbol{f}$ may contain more local variables than necessary, we need to remove the local variable assignments from $\boldsymbol{x}:=\boldsymbol{f}$. Given any $\boldsymbol{x}:=\boldsymbol{f}$ and a local variable set $\boldsymbol{l}$ then as long as such a multiple assignment is not bounded by a loop structure, we can always achieve $(\boldsymbol{x}-\boldsymbol{l}):=\boldsymbol{f}'$, where $\boldsymbol{f}'$ is the sub-set of $\boldsymbol{f}$ after removal of the expression set to which $\boldsymbol{l}$ corresponds. When $\boldsymbol{x}:=\boldsymbol{f}$ is in a loop structure, not all local variables can be removed, e.g., local variables may depend on their value in a previous iteration. The detailed treatment of this is given in [13].

After removing local variable assignments, either completely or partially, we can build a *Variable Dependency Directed Graph* (VDDG). The formal definition is:

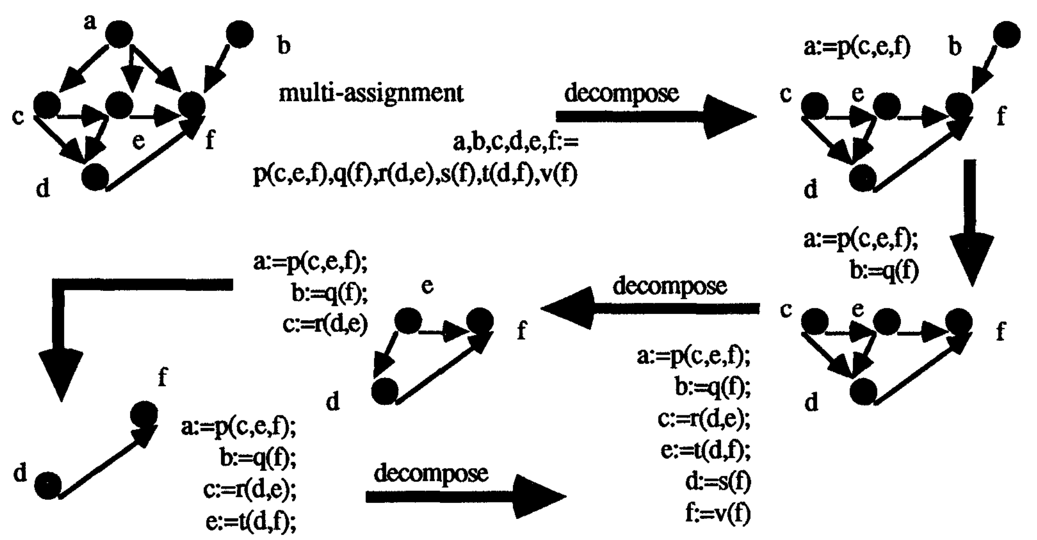
> Given any multiple assignment $\boldsymbol{x}:=\boldsymbol{f}$, its VDDG is a directed graph containing a node set $\boldsymbol{x}$ and an edge set $\{<x,y> \mid (x:=f)\in(\boldsymbol{x}:=\boldsymbol{f}),\ y\in(V(f)-\{x\})\}$, i.e., variable x connects to all its dependent variables except itself, where $V(f)$ is the variable set of f.

This graph cannot contain any self-cycles. The problem of determining the maximum number of local variables may be translated to determining the number of local variables that convert a multi-assignment into an implementable assignment sequence in an imperative program. Two cases apply for the VDDG. They are.

- the VDDG is a tree (or a number of trees) without any cycles
- the VDDG is a tree-structure (or a number of tree-structures) with cycle-nodes

When a given VDDG is a tree (or a number of trees), no local variable is needed because we can use an extended *Pre-Order* traversal of the tree to decompose the corresponding multi-assignment into a normal assignment sequence. This order also defines the principle for decomposing any directed graph with cycle-nodes. The following diagrams illustrate these ideas:

Decomposing a Multi-Assignment that Corresponds to a Tree into a Sequence:



When a VDDG is a tree-structure (or a number of tree-structures) with cycle-nodes then for a simple cycle that contains no subcycle a local variable is needed to decompose this cycle into a tree. However a more complex process [13] is needed to handle a complex cycle that contains nested subcycles.

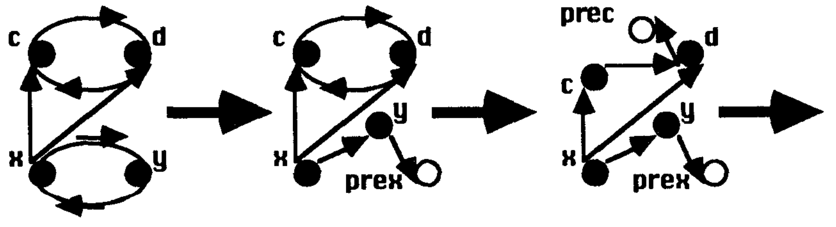
The three implementations below illustrate the impact of applying variable normalization. The normalized implementation (RaGCD) uses six rather than the seven or eight assignments used in the other two implementations. In Ra6CD each variable is used only for a single purpose.

Normalization of a Loop Body Sequence

Example:
Two implementations (ExGCD1 and ExGCD2) of the extended Euclid's GCD algorithm, have been presented by Jensen & Wirth[6] and Horowitz & Sahni[7] respectively:

```
ExGCD1:                        ExGCD2:                          RaGCD:
c:=M; d:=N; x:=0; y:=1;        c:=M; d:=N; x:=0; y:=1;          c:=M; d:=N; x:=0; y:=1;
do d≠0 →                       do ≠1 →                          do ≠0 →
    q:=c/d; r:=cmodd;              q:=c/d; r:=cmodd;                prex:=x; prec:=c;
    y:=y-q*x; c:=d; d:=r;          w:=x-q*y; c:=d; d:=r;            x:=y-c/d*x; y:=prex;
    r:=x; x:=y; y:=r               x:=y; y:=w                       c:=d;d:=precmodd
od;                            od;                              od;
return(y)                      if y<0 → y:=y+p fi; return(y)    return(y)
```

After applying equivalence transformations we get (q, r, c, d, x, y):=(c/d, x, d, c*mod*d, y-c/d*x, x) from the loop body of ExGCD1. After removal of local variables q and r, we obtain the underlying logical intent of the body, that is, (c, d, x, y) := (d, c*mod*d, y-c/d*x, x). Its VDDG and normalized result RaGCD are as follows:



```
prex:=x;
prec:=c;
x:=y-c/d*x;
y:=prex;
c:=d;
d:=prec mod d
```

5. Normalization of a Program Fragment

In this paper we have defined a set of normalization conditions for individual statements and for composite statements. These normal forms which have a mathematical/logical basis express an ideal. Taken together these requirements provide a basis for constructing what we will call *well-structured programs*. Such programs should be composed only of normalized statements at all levels. We claim that programs composed only of normalized components are far more likely to be less complex and more reliable. To make this point let us consider Manna's Abstract Program **AP** [10] as an example:

```
if p(y) → x:=y; do ¬p(x) → x:=f(x) od
[] ¬p(y) →
      x:=a;
      if p(x) → do ¬p(x) → x:=f(x) od
      [] ¬p(x) →
            x:=f(x);
            if p(x) → x:=a; do ¬p(x) → x:=f(x) od fi
      fi
fi
```

When this program fragment is normalized (and hence re-engineered) we get the following equivalent much simpler and non-redundant program fragment:

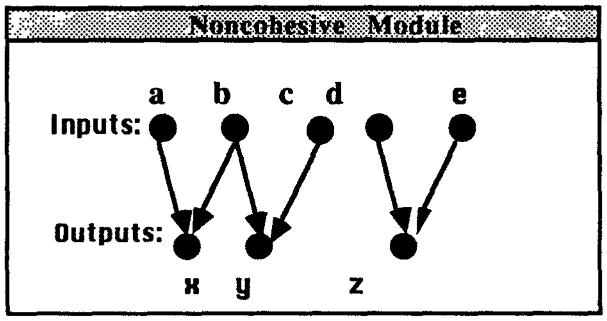
```
if p(y) → x:=y
[] ¬p(y) → if p(a) → x:=a [] ¬p(a) → x:=f(a) fi
fi
```

6. Module Normalization

Cohesion is usually cited as the criterion that may be used to decide what belongs in a given module and what should be implemented elsewhere[12]. Unfortunately this concept is rarely defined precisely enough to be used a basis for module normalization[5]. Two formal requirements must be satisfied for the functionality encapsulated in a module to be cohesive. These requirements maybe formulated in terms of input/output dependencies. Where mutual dependencies exist there is cohesion. Assessment of a module's cohesion is easiest to make using a graph-theoretic interpretation. The two normalization conditions are as follows:

MVN-1
Each output variable must depend upon at least one input, and each input variable must be used to produce at least one output variable.

MVN-2
The input/output variable dependencies must be represented by a *single* connected bipartite graph. The I/O dependencies in the diagram below violate this criterion .



If we denote the module input and output sets by **I** and **O**, and we construct the VDDG graph <I∪O, E>, where E is the edge set for the module/program then the normalization conditions may be expressed formally as follows:

Module Cohesion Conditions
MVN-1. $\forall x \exists y (x \in \mathbf{I} \wedge y \in \mathbf{O} \Rightarrow \langle x,y \rangle \in \mathbf{E})$ and
$\forall x \exists y (x \in \mathbf{O} \wedge y \in \mathbf{I} \Rightarrow \langle x,y \rangle \in \mathbf{E})$
MVN-2. $\forall \mathbf{I}' \forall \mathbf{O}' (\mathbf{I} \supset \mathbf{I}' \wedge \mathbf{O} \supset \mathbf{O}' \Rightarrow$
$\neg \exists \langle x,y \rangle \in \mathbf{E} (x \in \mathbf{I}' \wedge y \in \mathbf{O}-\mathbf{O}' \vee x \in \mathbf{I}-\mathbf{I}' \wedge y \in \mathbf{O}'))$

These two normalization criteria provide a very effective means for deciding what functionality belongs in a particular module. In the noncohesive module above the component with output z and dependent inputs d and e should be in a separate module.

7. Conclusions

Structured programming, information hiding and ADTs have made important contributions to program quality. However such techniques alone are often not strong enough to control (reduce) complexity to manageable levels, make software more reliable and make it easier to understand and change. The weakness of these methods when they are applied to structures is that they do not effectively take into account contextual and semantic issues. As a result, they do not guarantee that complexity will be limited and that high quality software will always be produced by following their guidelines.

What we have suggested in the present discussion is that there exists, beyond structured programming, another level of structuring, based on context and semantics. This level is more subtle, but it is significant in its impact on the quality of software.

In the past decades, much of the mathematics/formal methods that have been brought to bear on programs has looked only at the semantics/correctness of components. What we need to invent is *the mathematics/logic of structure for program components (composed and uncomposed)*. Normalization of simple and complex statements, of variable-usage and of modules provides a vehicle for achieving these goals. To realize the normal forms we have described we have three choices: get the programming language to enforce them, leave it to users to design components that conform to the normal form requirements, or develop tools that can automatically re-engineer un-normalized components to their corresponding normal forms. Obviously the first choice is the best. It does however require some refinements to existing languages and compilers.

References

1. Birkhoff, G., MacLane, S., *Algebra*, MacMillan, N.Y. 1967.
2. Dijkstra, E.W., Scholten, C.S.: *Predicate Calculus and Program Semantics*, Springer-Verlag, 1989.
3. Dahl, O.-J.,Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming*, Academic Press, N.Y., 1972
4. Dromey, R.G., *Program Derivation*, Addison-Wesley Press, 1989.
5. Dromey, R.G., *Cornering the Chimera*, IEEE Software, Jan., 1996.
6. Horowitz, E., Sahni, S., Fundamentals of Computer Algorithms, Computer Science Press Inc., 1978
7. Jensen, K., Wirth, N., PASCAL User Manual and Report, 2nd ed., Spring-Verlag,1974.
8. Kleene, S.C., *Mathematical Logic*, Wiley, N.Y., 1967.
9. Liskov, B.H., Zilles, S.N., *Programming with Abstract Data Types*, SIGPLAN Notices, 9(4), 50-59, Apr. 1974.
10. Manna, Z., *Properties of Programs and the First-Order Predicate Calculation*, JACM 16(2), 1969.
11. McDermid, J.A., Software Engineer's Reference Book, Butterworth-Heinemann, 1991.
12. Myers, G., *Software Reliability: Principles and Practices*, Wiley, NY (1976)
13. Pan, S., *Software Quality Improvement, Specification Derivation and Measurement Using Formal Methods*, PhD Thesis, Griffith Uni., Australia. 1995.
14. Pan, S., Dromey, R.G., Re-engineering Loops, Computer Journal (accepted for pub.)
15. Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules*, Comm. ACM., 5(12) 1053-1058, Dec. 1972.
16. Ullman, J.D., *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.