Automatic Derivation of Loop Termination Conditions to Support Verification

Daniel Powell

School of Information Technology
Griffith University, Gold Coast,
Queensland, Australia,
Email: d.powell@sqi.gu.edu.au

Abstract

This paper introduces a repeatable and constructive approach to the analysis of loop progress and termination conditions in imperative programs. It is applicable to all loops for which a variant function can be defined using only loop guard variables. The approach involves the algorithmic derivation of loop progress and termination conditions directly from the code itself. The derivation of these conditions has been automated in a prototype tool. The conditions yielded by the automated algorithms are useful for reasoning about correctness in verification based code inspections as well as for the documentation and assessment of program preconditions. Unlike existing formal approaches to termination investigation, which are reliant on the presence of formal specifications, this approach is applicable to undocumented programs as well as formally specified programs. We present the algorithms and formal methods implemented in a prototype tool for deriving loop progress and termination conditions and use the output generated by the tool to illustrate its use in supporting verification and termination defect correction.

Keywords: loop termination, automated formal methods, verification

1 Introduction

The verification of imperative programs involves proving a program satisfies its specification if it terminates, and proving that it terminates (Floyd 1967, Hoare 1969). A previous paper (Powell 2002*a*) dealt with the derivation of information to support human and mechanical reasoning about partial correctness. This paper deals with derivation of semantic information to support reasoning about loop progress and termination.

Techniques for proof of termination of imperative programs (Floyd 1967, Manna 1974, Alagic & Arbib 1978, Backhouse 1986, Dijkstra & Scholten 1989) are well understood. Two conditions must be met for a loop to terminate: (i) The loop must make progress toward establishing the termination condition described by the loop guard under any initial conditions allowed by the precondition, and (ii) the loop guard must be strong enough to force termination after a finite number of iterations. This paper describes a mechanised approach to yielding loop progress and termination conditions. These conditions are useful to support reasoning about termination.

Existing techniques for investigating termination are generally based on investigating conditions on a suitable *variant function*(Alagic & Arbib 1978) or on the application of the *weakest precondition predicate transformer*(Dijkstra & Scholten 1989). Weakest precondition techniques are primarily used for program derivation. For this reason the majority of comparative discussion in this paper concerns only variant function techniques. We propose a constructive technique that yields loop progress and termination conditions directly from code, without reliance on the generation or provision of a variant function. This is achieved by applying the strongest postcondition predicate transformer (*sp*)(Back 1988, Dijkstra & Scholten 1989, Gannod & Cheng 1996, Powell 2002*b*) to the code under a precondition (*true* if none is specified) to yield a specification to which heuristics are applied to investigate properties that must hold if progress is to be achieved and termination guaranteed. The calculation of the strongest postcondition has been sufficiently automated(Powell 2002*a*, Powell 2002*b*) to allow these heuristics to be mechanically applied.

As an example, a prototype[1] implementing the techniques presented in this paper, yields the progress and termination conditions in Figure 1 for the following loop intended to sum every second array element in an array from element 0 to n.

```
{true}
procedure sum2 (a : seq Int, n : Int, s : Int ) {

    i := 0 ;
    s := 0 ;
    do i ≠ n →
        i := (i + 2) ;
        s := (s + a[i])
    od

}
```

The conditions presented in Figure 1 are useful for reasoning about termination. In this case the generated report states that the loop will only progress toward termination when $i < n$ and terminate when n? (the actual parameter value for the formal parameter n) is divisible by 2. The program precondition *true* does not ensure that n is a positive number divisible by 2. The procedure is, therefore, reasoned to be defective. This approach to reasoning about termination is constructive. That is, we are now armed with information that allows us to either strengthen the precondition or modify the loop guard to correct the defect.

This paper presents theorems and heuristics for generating these termination and progress conditions and discusses their use in supporting verification and correcting resulting defects. The approach presented in this paper is applicable wherever variant function

[1] The prototype architecture is discussed in detail in (Powell 2002*b*). The prototype produces a report in LaTeX format. Any tool output, referred to in this paper has been cut and paste directly from this generated output.

For the loop

```
i := 0;
do i ≠ n →
      i := (i + 2) ;
      s := (s + a [i])
od
```

We calculate the iterative-form invariant

$$R_{k+1} \equiv \left(\begin{array}{l} a_{k+1} = a_k \wedge n_{k+1} = n_k \wedge i_k \neq n_{k+1} \wedge k \in \mathbf{N} \wedge i_{k+1} = (2 + i_k) \wedge \\ s_{k+1} = (s_k + a\,[i_{k+1}]) \end{array} \right)$$

Loop Progress Conditions
The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$(\ (i < n)\ )$$

Loop Termination Conditions
The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N}\,(2 \times k = n?)$$

Figure 1: Loop progress and termination conditions yielded by the tool for `sum2`

techniques can be applied. The remainder of this paper is organised as follows. Section 2 provides formal methods and specific notational background and discusses related variant function techniques for investigating loop termination. Section 3 presents a technique for deriving loop progress conditions and presents small examples with reports generated by the prototype tool. In section 4 we present rules and a theorem for investigating termination satisfiability. Section 5 discusses conclusions and future directions of this research. The generation of loop progress and termination conditions has been implemented in a prototype tool. Relevant sections of the reports generated by the tool are presented throughout this paper for the worked examples.

2 Preliminaries

Examples are presented in this paper using a syntax resembling Dijkstra's Guarded Commands Language (Dijkstra 1975).

Substitution
Given any formula F, $F[y/x]$ denotes the formula obtained from F by replacing all free occurrences of x in F by y, where x and y are expressions.
For example, $(x = y + b \wedge y = a)\,[a/y] \equiv (x = a + b \wedge a = a) \equiv (x = a + b)$.
We extend the substitution function to operate on an expression e, such that $e[y/x]$ denotes the expression e with all free occurrences x replaced by y.

The Free Variable Function
The free-variable function, $\mathbf{V}$, takes as its argument, F, a formula, term or program, and returns the set of free variables contained in F, such that:
$\mathbf{V}(F) = \{x | x$ occurs in F and x is a free variable$\}$
For example, $\mathbf{V}(x > 3 \wedge y \leq x) = \{x, y\}$.

Notation: Parameter Notation
For any formal procedure parameter, p, we use $p?$ to denote the actual parameter value of p at invocation of the procedure.

Notation: Iteration Dependence
For any variable, v, v_k represents the value of v on the kth iteration of a loop. v_0 represents the value of v on initialisation of a loop. For any formula, F, F_k represents the value of F on the kth iteration of a loop. F_0 represents the formula F on initialisation of a loop. $F_{i,k}$ represents F on the kth iteration of a loop executing through path i of the loop body, where there is more than one path through the loop. For any function, f, f_k represents the value of f on the kth iteration of a loop. f_0 represents the value of f on initialisation of a loop. $f_{i,k}$ represents the value of f on the kth iteration of a loop executing through path i of the loop body, where there is more than one path through the loop.
That is, given any formula or function, P, $P_k \equiv P\,[v_k/v]$, for all variables $v \in \mathbf{V}(P)$.

Difference Equations
An equation which expresses a value of a term in a sequence as a function of other terms in the sequence is called a *difference equation* or *recurrence relation* in iterative form. An equation which expresses the value of a term v_{k+1} of the sequence $\langle v_n \rangle$ in terms of the value of the term v_k is called a *first-order difference equation*. If for a given difference equation, we can determine an explicit formula for v_k, where $k \geq 0$, in terms of v_0, then we can solve the difference equation. If such a formula exists, then we will call it the *solution* to the difference equation and denote it $v(k)$. For example, assuming $v_0 = 0$ and having $v_{k+1} = v_k + 1$. then $v(k) = v_0 + k = k$. So, $v_{k+1} = v(k + 1) = k + 1$.

It is assumed that readers are familiar with first-order logic. If they are not, we direct them to (Dromey 1989) for a good discussion on logic for program design.

2.1 The strongest postcondition predicate transformer

In order to automate the derivation of loop progress and termination conditions from code, a mechanical technique for deriving semantics from individual statements is needed. To provide the required mapping from syntax to semantics, the predicate calculus *strongest postcondition predicate transformer* (Back 1988, Dijkstra & Scholten 1989, Gannod & Cheng 1996, Powell 2002*b*) is used. The following definitions for *sp* have been implemented in the prototype tool:

- *sp* for assignment (Powell 2002*b*)

$$sp(Q,\ \texttt{x} := \texttt{E}) \equiv (Q[v/x] \wedge x = E[v/x])$$

where $Q \Rightarrow x = v$. If x is not initialised prior to assignment, then we let $v = x_0$.

- *sp* for selection (Dijkstra & Scholten 1989)

$$\begin{array}{rl} & sp\,(Q,\ \texttt{if}\ C_1 \rightarrow S_1\ []\ \ldots\ []\ C_n \rightarrow S_n\ \texttt{fi}) \\ \equiv & sp\,(Q \wedge C_1,\ S_1) \vee \ldots \vee sp(Q \wedge C_n,\ S_n) \end{array}$$

- *sp* for iteration (Dijkstra & Scholten 1989, Pan 1994)

$$sp(Q,\ \texttt{do}\ G \rightarrow S\ \texttt{od}) \equiv \neg G \wedge (Q \vee P_1 \vee \ldots \vee P_m)$$

where m is a positive integer representing the number of iterations, and $P_1 = sp(Q \wedge G,\ S)$ and $P_{i+1} = sp\,(P_i \wedge G,\ S)$ for $0 < i \leq m$.

- *sp* for sequential composition (Dijkstra & Scholten 1989)

$$sp\,(Q,\ S_1; S_2) \equiv sp\,(sp\,(Q,\ S_1)\,,\ S_2)$$

The application of the definition given above for calculating *sp* for iterative constructs is difficult, if not impossible, in practice, because, although the calculation depth, m, is bound, it is not fixed. We avoid this problem by deriving an *iterative-form invariant* describing the state changes of all accessible variables on any arbitrary iteration of a loop.

2.2 The iterative-form invariant

The iterative-form loop invariant describes the state changes resulting from a single, arbitrary $k+1$st iteration of a loop ($k \geq 0$). It specifies values after a $k+1$st iteration for all variables in the variable set of the loop initialisation, loop body and loop guard.

To model state change on an arbitrary iteration a description of initial state must be provided. The description of initial states can be derived directly from the code. For any loop, $\texttt{do}\ G \rightarrow S\ \texttt{od}$, executing under the precondition Q, we denote the weakest conditions required for a $k+1$st iteration by Q_{k+1} and calculate as

$$Q_{k+1} = (k \in \mathbb{N} \wedge G_k \wedge v_{k+1} = v_k \wedge C) \qquad (1)$$

for all free-variables $v \in \mathbf{V}(Q) \cup \mathbf{V}(S) \cup \mathbf{V}(G)$, and where C denotes the conjunction of all predicates in Q involving only constants or variables not changed by G or S. By definition, a predicate involving only constants or variables not changed by G or S is invariant and must, therefore, hold at the beginning of any arbitrary iteration. The condition Q_{k+1} states that the values of the loop guard variables immediately prior to the $k+1$st iteration (the values immediately after the kth iteration - initialisation if $k = 0$) satisfy the guard. Q_{k+1} also states that the values of all program variables remain unchanged between the end of the kth iteration (initialisation if $k = 0$) and the commencement of the $k+1$st iteration. This definition only applies for loops where the loop guard has no side effects in terms of updating program variables. Loop guards involving procedures with side effects are dealt with in (Powell 2002*b*). The ability to deal with such guards has not yet been implemented in the prototype tool, and is not addressed in this paper.

An iterative-form loop invariant can be calculated by application of the strongest postcondition predicate transformer. For any loop $\{Q\}\ \texttt{do}\ G \rightarrow S\ \texttt{od}$, we calculate the iterative-form invariant, denoted R_{k+1}, on an arbitrary $k+1$st iteration by

$$R_{k+1} = sp\,(Q_{k+1}, S) \qquad (2)$$

It can be considered that all statements in S when executed on an arbitrary $k+1$st iteration involve only $k+1$st iteration variable instances. According to the notation presented above, a reference to a variable v in the loop body S on a $k+1$st iteration is denoted v_{k+1}. In order to automate the calculation of (2) using the existing definitions for *sp*, we need to actually implement the rule

$$R_{k+1} = sp\,(Q_{k+1}, S\,[v_{k+1}/v]) \qquad (3)$$

for all variables $v \in \mathbf{V}(S)$.

Consider the following simple array initialisation program:

```
{a ∈ seq ℕ ∧ N > 0}
i := 0;
do i < N →
     a[i] := i;
     i := i+1;
od
```

The loop precondition is calculated as

$$Q \equiv (a \in \textbf{seq}\ \mathbb{N} \wedge N > 0 \wedge i = 0)$$

The initial conditions on an arbitrary $k+1$st iteration, calculated by (1), are

$$Q_{k+1} \equiv (k \in \mathbb{N} \wedge i_k < N \wedge i_{k+1} = i_k \wedge a_{k+1} = a_k \wedge N > 0)$$

Now, R_{k+1}, the iterative form invariant after the $k+1$st iteration, by (3), is

$$\begin{array}{rl} & R_{k+1} \\ \equiv & sp\,(Q_{k+1},\ \texttt{a[i]:=i;i:=i+1}) \\ \equiv & sp\left(Q_{k+1}, \texttt{a}\,[\texttt{i}_{k+1}]_{k+1} := \texttt{i}_{k+1}; \texttt{i}_{k+1} := \texttt{i}_{k+1}+1\right) \\ \equiv & \left(\begin{array}{l} k \in \mathbb{N} \wedge N > 0 \wedge i_k < N \wedge \\ a\,[i_k]_{k+1} = i_k \wedge i_{k+1} = i_k + 1 \end{array}\right) \end{array}$$

The iterative-form invariant forms the basis of the loop progress and termination investigation techniques presented in this paper as well as the basis for verification and invariant generation techniques presented in (Powell 2002*a*) and (Powell 2002*b*).

2.3 Related Work

The work most closely related to this approach involves the use of variant functions for investigating conditions for loop termination (Floyd 1967, Manna 1974, Alagic & Arbib 1978).

Variant Function techniques are based on *well-founded sets* (Manna 1974). A well-founded set is a partially ordered set with no infinite decreasing sequences. In a proof of termination, the verifier is required to identify a variant function, t, which maps program variables onto the well-founded set. The set of natural numbers with the ordering, $<$, is often chosen as this well-founded set. The variant function t, is suitable when t evaluated at the start of any iteration maps to a natural number that is greater than t evaluated at the end of the iteration, for every iteration in which progress is made toward termination. Loops fail to terminate when the statements in the loop body fail to decrease the variant function, or when the loop initialisation, loop guard and loop body imply that the variant is not bound to a well-founded set. In summary, loop progress and termination can be investigated by exploiting the formal properties

of a variant function. This technique is reliant on the provision or calculation of a suitable variant function.

Denoting the variant function by t, where $t : \mathbb{P}\ V(G) \rightarrow \mathbb{N}$, t_k represents the function t evaluated at the end of a kth iteration.

Expanding on the termination rules of (Floyd 1967, Manna 1974, Alagic & Arbib 1978), we can define rules for progress based on an identifiable variant function, t as follows:

$$t_k, t_{k+1} \in W \wedge t_{k+1} < t_k \tag{4}$$

where W is the well-founded set $(\mathbb{N}, <)$, and $k \in \mathbb{N}$.

The variant conditions for loop progress can be stated more simply as

$$0 \leq t_{k+1} < t_k, \text{ where } k \in \mathbb{N}. \tag{5}$$

This states that the variant function, t, evaluates to a natural number greater than zero at the start of every iteration of a loop and decreases with every iteration, but never below 0.

Given a suitable variant function, loop progress and termination can be investigated by evaluating the variant conditions (5) under R_{k+1}. This approach to deriving loop progress and termination conditions is described in detail in (Powell 2002*b*). It has the disadvantage that a suitable variant function is required to be either generated or provided. In practice, a suitable variant function is often not provided and the generation of variant functions is non-trivial, requiring the derivation of a loop invariant to determine suitability (Manna 1974, Powell 2002*b*). This is due to the fact that for a variant function to be suitable, it must be implied by the invariant. If this can't be proved, then a verifier can not be sure that the variant function being used for termination analysis is suitable. The remainder of this paper discusses an approach to investigating loop progress and termination in the absence of a variant function.

3 Investigating Loop Progress Without a Variant

In this section, we present a mechanisable technique for deriving loop progress conditions that is not reliant on the provision or calculation of a variant function.

The theory is introduced with simple examples where the loop guards are literals[2]. We extend this approach to deal with more complex guards at the end of this section.

For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, where G is a literal, $\neg G$ can be written as an equality $v = w$, where v and w are expressions possibly involving the addition of an introduced constant. For example, the negation of the loop guard $i < j$ $(i \geq j)$ can be written as $i = j + D$, where $D \geq 0$, which is in the form $v = w$. We introduce a constant, C, such that on termination $C = v = w$ and state that loop can progress toward this state with every iteration only when the difference between v and C or w and C decreases with each iteration of the loop.

For a loop with termination condition $C = v = w$, where C is a constant and v and w are expressions involving only loop guard variables and introduced constants, we present the possibilities for loop progress on a number line in Fig. 2.

The sub-figures in Fig. 2 are annotated with predicates that describe changes to v and w that must be possible on successive, arbitrary, iterations of a loop, if the corresponding loops are to make progress. For example, Fig. 2(c) states that both v and w must potentially change with every iteration. If the changes to v and w occur in two different paths through the loop body, then both paths must be reachable on an arbitrary iteration. At least one of these possible changes must actually occur on every iteration if progress is to be made toward termination on every iteration.

Pan and Dromey(Dromey & Pan 1996) present a normal form for loops known as a Multi-Branching Statement (MBS) form, in which all branches are accessible on every iteration (Dromey & Pan 1996). We assume that loops have been normalised to this form so that if progress toward termination is to occur, then at least one path through the body of the loop should ensure progress is made on any iteration.

Analysing further the diagrammatic representation of loop progress in Fig. 2 we identify the following conditions that must be met if progress is to be possible on every iteration of a loop(Powell 2002*b*). Cancelling out the references to the constant C, the loop body must ensure that *one* of the conditions in Figure 3 holds on any iteration in order for progress to be achieved:

$$\begin{aligned}
PC_1 &= (v_k < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} < w_k) \\
PC_2 &= (w_k < v_k \wedge w_k < w_{k+1} \wedge v_{k+1} < v_k) \\
PC_3 &= (v_k < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} = w_k) \\
PC_4 &= (w_k < v_k \wedge w_k < w_{k+1} \wedge v_{k+1} = v_k) \\
PC_5 &= (v_k < w_k \wedge v_k = v_{k+1} \wedge w_{k+1} < w_k) \\
PC_6 &= (w_k < v_k \wedge w_k = w_{k+1} \wedge v_{k+1} < v_k) \\
PC_7 &= \left(\begin{array}{l} v_k < w_k \wedge v_{k+1} \leq v_k \wedge w_{k+1} < w_k \wedge \\ v_k - v_{k+1} < w_k - w_{k+1} \end{array} \right) \\
PC_8 &= \left(\begin{array}{l} w_k < v_k \wedge w_{k+1} \leq w_k \wedge v_{k+1} < v_k \wedge \\ w_k - w_{k+1} < v_k - v_{k+1} \end{array} \right) \\
PC_9 &= \left(\begin{array}{l} v_k < w_k \wedge v_k < v_{k+1} \wedge w_k \leq w_{k+1} \wedge \\ w_{k+1} - w_k < v_{k+1} - v_k \end{array} \right) \\
PC_{10} &= \left(\begin{array}{l} w_k < v_k \wedge w_k < w_{k+1} \wedge v_k \leq v_{k+1} \wedge \\ v_{k+1} - v_k < w_{k+1} - w_k \end{array} \right)
\end{aligned}$$

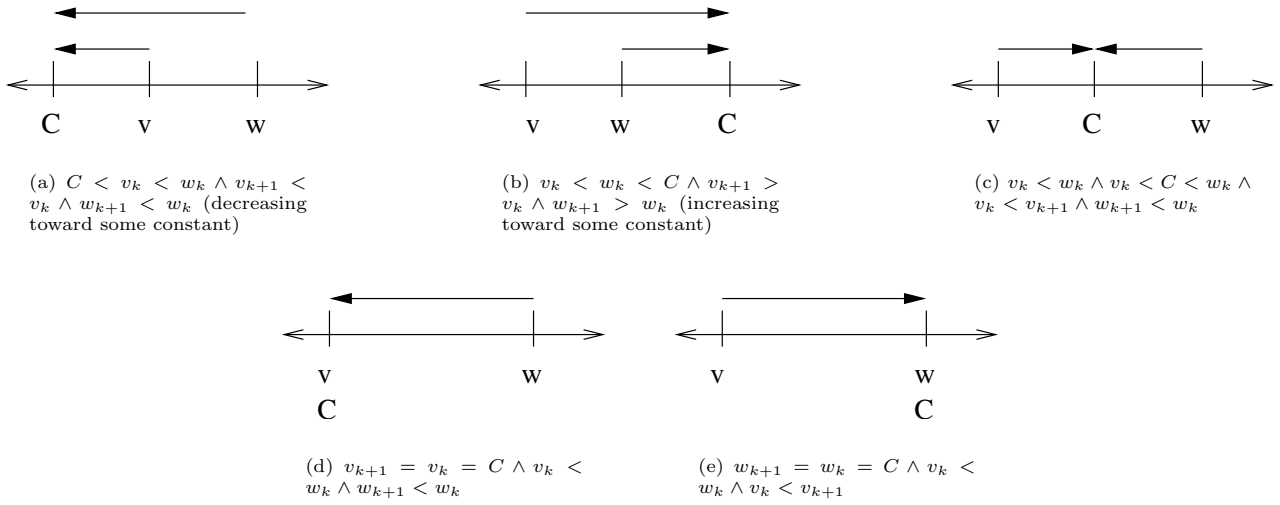
Figure 3: Required Loop Progress Conditions

The conditions defined in Fig. 3 rely on the comparison of variables on successive loop body iterations. We use the iterative-form invariant, R_{k+1}, to evaluate the progress conditions for a loop. The iterative-form invariant describes the relationship between $k+1$st and kth iteration instances of all variables yielded from each path through the loop body. In order to establish the initial conditions required for loop progress, denoted PC, we can simplify the following for loop with $N > 0$ paths and an iterative-form invariant R_{k+1}:

$$PC = R_{k+1} \wedge \bigvee_{i=1}^{10} (PC_i) \tag{6}$$

This produces a disjunctive predicate with references to both $k+1$st and kth iteration variable instances. As we wish to discover only the conditions required for progress at the beginning of any iteration, we attempt to substitute out all references to the $k+1$st iteration variable instances. This is simple due to the nature of the iterative-form invariant, R_{k+1}. For every variable, x, in the loop guard, loop

[2]We only consider literals constructed with the predicate symbols $<, \leq, =, >, \geq$ and their negations. Guards constructed with $\neq$ can only be handled by the techniques presented in section 4.



(a) $C < v_k < w_k \wedge v_{k+1} < v_k \wedge w_{k+1} < w_k$ (decreasing toward some constant)

(b) $v_k < w_k < C \wedge v_{k+1} > v_k \wedge w_{k+1} > w_k$ (increasing toward some constant)

(c) $v_k < w_k \wedge v_k < C < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} < w_k$

(d) $v_{k+1} = v_k = C \wedge v_k < w_k \wedge w_{k+1} < w_k$

(e) $w_{k+1} = w_k = C \wedge v_k < w_k \wedge v_k < v_{k+1}$

Figure 2: Possible state changes for loop progress such that on termination $v = w = C$.

body and loop precondition, there will be a corresponding equality in each disjunct of R_{k+1} of the form $x_{k+1} = E$, where E is an expression. These equalities are, therefore, maintained in each of the disjuncts of PC calculated by (6). We can remove all references to $k+1$st iteration variable instances from the progress conditions, PC, by the substitution

$$PC_i\left[E/x_{k+1}\right], \text{for all } x \in \mathbf{V}(Q) \cup \mathbf{V}(G) \cup \mathbf{V}(S). \tag{7}$$

where $PC_i \implies x_{k+1} = E$, for each disjunct, PC_i, of PC.

The iteration count k has been selected arbitrarily, and so PC after (7) defines the progress conditions at commencement of any $k+1$st iteration (by the law of *universal generalisation*). As there are no references to any variable instances other than kth iteration instances and constants, we can make PC more readable by dropping the variable instance subscripts and state that PC must hold at commencement of any iteration for progress to occur. That is, for progress to occur the satisfiability of PC at the commencement of any iteration is invariant.

To summarise, we have the following theorem.

Theorem 1. *[Progress Conditions Method] For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, apply the following steps:*

1. *Calculate the iterative form invariant by (2).*
2. *Identify expressions v and w such that on termination $v = w$.*
3. *Calculate the progress conditions, PC, by (6).*
4. *Simplify the progress conditions, by applying (7) to substitute equal expressions for all $k+1$st iteration variable instances. Drop all k subscripts from the simplified progress conditions.*
5. *Calculate the failure to progress conditions, FP by negating PC.*

Then, PC describes the conditions required for loop progress on any iteration, FP describes the conditions under which the loop will fail to progress toward termination. If the conjunction $Q \wedge FP$ is satisfiable then there are initial conditions allowable under the given precondition that will result in S failing to progress toward termination. The states satisfying $Q \wedge FP$ describe these initial conditions. Conversely, the progress conditions PC, conjoined to the precondition Q, describes the required initial conditions in order for S to progress toward achieving termination with every iteration. $Q \wedge PC$ is, therefore, the weakest precondition, and PC is the weakest invariant required for loop progress on every iteration. Conditions for progress or termination, TC, are defined by $TC = PC \vee \neg G$.

Corollary 1. *Any failure to terminate defects can be corrected in one of the following ways: (1) By strengthening the precondition from Q to $Q \wedge (\neg G \vee PC)$, if acceptable, (2) By modifying S so that progress is possible under the precondition Q, or (3) By strengthening the guard to $G \wedge PC$.*

As implied by corollary 1, not only are the derived conditions useful for reasoning about correctness, they are also constructive in that they can be used to support the correction of loop progress related defects.

Defect Correction by Strengthening the Precondition. Theorem 1 states that $Q \wedge PC$ is the weakest precondition, and PC is the weakest invariant required for loop progress on every iteration. Based on this, we can correct any failure to progress defects by strengthening the precondition to $Q \wedge PC$. To ensure that the precondition implies that either progress will be achieved or termination achieved we can strengthen the precondition to $Q \wedge (\neg G \vee PC)$. For languages that allow assertions, we can always ensure progress by asserting PC as an invariant in the body of the loop. These program contract techniques for defect correction are not always desirable as they may result in a precondition that is too strong to be useful.

Defect Correction by Modifying the Loop. In some cases the given precondition may be correct although the derived failure to progress conditions are allowed by it. In these cases, the loop itself must be defective. That is, the loop guard or loop body statements are responsible for failure to progress and terminate. There are basically two ways in which we can modify the loop itself to progress under the given precondition.

(1) We can strengthen the loop guard. For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, strengthening the guard to $G \wedge PC$, will ensure that progress is always made by S under the precondition Q. This is logically the same as asserting PC as an invariant at the beginning

of each iteration. (2) In many cases, while strengthening the guard will ensure termination, it may not ensure the correct invariant is maintained, and therefore it may cause the loop to terminate with an incorrect postcondition. In such cases, we need to modify the body of the loop itself and possibly also the guard, so that progress may be achieved while maintaining the correct invariant. The derived loop progress conditions may be of benefit in guiding the re-engineering of such loops.

As an example of applying the *Progress Conditions Method*, consider the Greatest Common Divisor example in Program 1.

Program 1 GCD Example

```
procedure gcd(x,y:  INOUT ℤ) {
    do x ≠ y →
        if x > y →
           x := x-y;
        [] y > x →
           y := y-x;
        fi;
    od
}
```

By the *Progress Conditions Method* we derive the loop progress conditions.

Step 1: Calculate the iterative form invariant
We assume the weakest possible precondition *true*, ensuring that the loop precondition is $(x, y \in \mathbb{Z} \wedge x = x? \wedge y = y?)$. Applying (1) then (2) to the loop body code yields the iterative form invariant

$$R_{k+1} \equiv \begin{array}{l} k \in \mathbb{N} \wedge \\ \left(\begin{array}{l} x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \vee \\ x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right) \end{array}$$

Step 2: Identify expressions v and w such that on termination $v = w$
The negation of the loop guard states that on termination the equality $x = y$ holds, so we identify and substitute x for v and y for w.

Step 3 and 4: Calculate the progress conditions, PC, and simplify by (7)

$$\begin{array}{lll} PC & \equiv & R_{k+1} \wedge \bigvee_{i=1}^{10} (PC_i) \\ & \equiv & \begin{array}{l} k \in \mathbb{N} \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} \bigvee_{i=1}^{10} (PC_i) \wedge \\ x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \end{array} \right) \vee \\ \left(\begin{array}{l} \bigvee_{i=1}^{10} (PC_i) \wedge \\ x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right) \end{array} \right) \end{array} \\ & \equiv & k \in \mathbb{N} \wedge (0 < y_k < x_k \vee 0 < x_k < y_k) \\ & \equiv & k \in \mathbb{N} \wedge (x_k \neq y_k \wedge 0 < x_k \wedge 0 < y_k) \end{array}$$

Dropping the k subscripts gives the following, more natural, representation of the progress conditions:

$$PC \equiv x \neq y \wedge 0 < x \wedge 0 < y$$

So, whenever the guard is not negated $(x \neq y)$ and x and y are both greater than zero at the start of any iteration, the loop will make progress toward termination on that iteration.

This approach has been automated in the prototype. The progress conditions report generated for this example is shown in Fig. 4. As can be seen, the prototype, even with its basic simplification rules, manages to reduce the twenty disjuncts of $R_{k+1} \wedge \bigvee_{i=1}^{10} (PC_i)$ to a form in which the simplification required is trivially applied manually.

Step 5: Calculate the failure to progress conditions, FP.
The failure to progress conditions are the negation of the progress conditions. So,

$$\begin{array}{lll} FP & \equiv & \neg PC \\ & \equiv & x = y \vee x \leq 0 \vee y \leq 0 \end{array}$$

Whenever either the guard is negated $(x = y)$, or either of x or y are less than or equal to zero at the start of any iteration then the loop will fail to make progress toward termination on that iteration. The negation of the guard is not of concern as the loop is expected to not progress on termination, however, the other two states must be considered.

In order for progress to be made on an iteration, the progress conditions state that $x \neq y \wedge x > 0 \wedge y > 0$ must hold. So, $x \neq y \wedge x > 0 \wedge y > 0$ is an invariant required for loop progress. It can be reasoned that if $x > 0 \wedge y > 0$ is maintained initially then it will continue to be maintained under any iteration of the loop body and on termination.

The loop initialisation under the weakest precondition *true* yields the loop precondition $(x = x? \wedge y = y? \wedge x, y \in \mathbb{Z})$. Reasoning about correctness, we note that this precondition does not exclude the states described by FP. Therefore, the loop may be entered with initial values for x and y that will cause failure to progress.

Corollary 1 says that we can use the derived progress conditions to correct the defect causing nontermination by strengthening the precondition. In this situation the defect is not in the program but in the program's specification. By strengthening the precondition to

$$(x = x? \wedge y = y? \wedge x, y \in \mathbb{Z} \wedge (x = y \vee x > 0 \wedge y > 0))$$

we reason that the loop will always progress toward termination. ■

3.1 Loops with Complex Guards

So far we have only discussed the analysis of loops with simple, literal guards. In order to apply these methods to any real examples, it must be applicable to loops with complex guards. Here we define a complex guard as being a conjunction or disjunction of quantifier free formulae.

Theorem 2 (Complex Guards). *For any loop with a complex guard, G, for each literal, p_i, in $\neg G$, apply the method described in this section to derive the progress conditions for that part of the guard. If the progress conditions for p_i are denoted by PC_i, then the progress conditions for the entire loop will be given by*

$$PC \equiv \neg G\,[PC_i/p_i]\,, \text{ for all } p_i \text{ in } G \qquad (8)$$

Proof. If the negated guard is conjunctive, progress must be made toward achieving each conjunct. That is, given a negated guard of the form $p_1 \wedge \ldots \wedge p_n$, let $PC_1 \ldots PC_n$ be the conditions required for progress toward achieving $p_1 \ldots p_n$ respectively. Then, in order for the loop to progress toward termination the loop must progress toward achieving all of the conjuncts in the conjunctive termination condition. If any p_i is not satisfiable, then progress may not succeed. So, the conditions required for progression toward a conjunctive termination condition are the conjunction of the individual conjunct's progress conditions, $PC_1 \wedge \ldots \wedge PC_n$.

Loop Progress Conditions The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\begin{pmatrix} (x < y \wedge 0 < x) \vee \\ (y < x \wedge 0 < y) \end{pmatrix}$$

Loop Termination Conditions The loop will terminate, iff the following is satisfiable[1]:

$$\exists k \in \mathbb{N} \, (x_k = y_k)$$

[1]Assuming that the loop progress conditions are satisfied and the loop precondition ($x = x? \wedge x \in \mathbf{Int} \wedge y = y? \wedge y \in \mathbf{Int}$) holds, does R_{k+1} imply that the termination conditions are satisfiable?

Figure 4: Loop progress and termination conditions yielded by the tool for `gcd`

If the negated guard is disjunctive, progress must be made toward achieving at least one of the disjuncts of the negated guard. That is, given a negated guard of the form $p_1 \vee \ldots \vee p_n$, let $PC_1 \ldots PC_n$ be the conditions required for progress toward achieving $p_1 \ldots p_n$ respectively. Then, in order for the loop to progress toward termination the loop must progress toward achieving at least one of the disjuncts in the disjunctive termination condition. If no p_i is achieved then termination will not succeed. So, the conditions required for progression toward a disjunctive termination condition are the disjunction of the individual disjunct's progress conditions, $PC_1 \vee \ldots \vee PC_n$.

This is recursive for each sub-formula of the negated guard. □

For example, consider the following:

$\{i = I \wedge j = J\}$
`do i≠100 ∧ j≠0 →`
`  i:=i+1; j := j-1;`
`od;`

The termination condition is $i = 100 \vee j = 0$ The conditions for progress toward achieving $i = 100$ are derived as $i < 100$. The conditions for progress toward achieving $j = 0$ are derived as $j > 0$. Therefore in order for the loop to progress toward termination, either $i < 100$ must hold, or $j > 0$ must hold, on every iteration. If this is not the case progress is made toward neither disjunct of the termination condition.

By (8), the progress conditions are

$$i < 100 \vee j > 0.$$

The failure to progress conditions are, therefore,

$$i \geq 100 \wedge j \leq 0.$$

The given loop precondition does not exclude these states, so it is possible to begin this loop with initial values that will cause progress failure.

4 Investigating Termination Conditions

In order to completely analyse loop termination, we not only need to reason about loop progress, but also that the guard will at some stage force termination. That is, termination must be investigated based on the principle that in order for a loop to terminate, at some stage the loop guard must be negated.

Consider any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`. In order for termination to be achieved, after some arbitrary kth iteration, where $k \in \mathbb{N}$, $\neg G_k$ must hold. This can be described formally by (9).

$$\exists k \in \mathbb{N} \, (\neg G_k) \tag{9}$$

In order to evaluate this condition, it is necessary to determine if the loop guard variables can ever be in a state on any single iteration, k, which ensures G is negated.

On some single iteration every program variable in the loop guard must be instantiated with a value that will make the loop guard false, forcing termination. If this can not occur, then termination will not be achieved. If, based on iterative-form difference equations in R_{k+1}, a difference equation solution, $v(k)$, exists for all variables, v, in the variable set of the loop guard, G, then the conditions required for termination can be defined by simplifying (10).

$$\exists k \in \mathbb{N} \, (\neg G \, [v(k)/v]) \, , \text{ for all } v \in V(G) \tag{10}$$

That is, there must be an iteration k, such that the value of every loop guard variable on that iteration forces the guard to be negated.

The method just described is summarised in the following theorem.

Theorem 3. *[Guard Substitution Method]*
For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, apply the following steps:

1. *Calculate the iterative form invariant by (2).*
2. *Calculate difference equation solutions[3] for all variables in the variable set of the loop guard from the equalities in the iterative form difference equation.*
3. *Using these difference equation solutions, solve (10) for $k \in \mathbb{N}$.*

If (10) can't be solved for $k \in \mathbb{N}$, then the loop will not terminate and $\lceil k \rceil$ is the iteration on which progress toward termination ceases. If (10) can be solved for $k \in \mathbb{N}$, then the loop will eventually terminate. The k that satisfies (10) is the number of iterations to termination.

Proof. From the discussion presented above, if (10) can not be solved for any natural number iteration count k, then there is no single iteration in which the variables involved in the loop guard will be instantiated with values that will force termination. Because k was chosen arbitrarily, we can generalise and say that termination will never be forced in these situations. Conversely if (10) can be solved for at least one natural number k, then there must be at least one iteration k in which the loop guard variables are instantiated with values which negate the loop guard forcing termination. □

Corollary 2. *For any loop, if (10) can be solved for k ensuring $k \in \mathbb{N}$, then k is a direct measure of the order complexity of the loop.*

[3]The prototype tool can solve a number of forms of difference equations which commonly occur in loops. These difference equation solutions are defined in (Powell 2002*b*).

Loop Termination Conditions
The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N} \left(\left(0 + \sum_{j=0}^{k} (j) \right) = 100 \right)$$

Figure 5: Loop termination conditions yielded by the tool for Program 2

Program 2 Sum Example

```
{true}
i:=0; s:=0;
do s ≠ 100 →
    i:=i+1;
    s:=s+i;
od
```

Consider the example loop in Program 2. We apply theorem 3 to Program 2 to investigate termination.

Step 1: Calculate the iterative-form invariant, R_{k+1}
The given precondition asserts ($true$). This makes the loop precondition ($i = 0 \wedge s = 0$). The negation of the guard is $s = 100$. We calculate the iterative form invariant, R_{k+1} as

$$R_{k+1} \equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge \\ s_{k+1} = s_k + i_{k+1} \wedge s_k \neq 100 \end{array} \right)$$

Step 2: Calculate difference equation solutions for all loop guard variables
The only loop guard variable is s. The assignment to s, yields the difference equation $s_{k+1} = s_k + i_{k+1}$. The assignment to i, yields the difference equation $i_{k+1} = i_k + 1$. The solution to this difference equation, is $i(k) = k$. So, the solution to $s_{k+1} = s_k + i_{k+1}$ is

$$\begin{aligned} s(k) &= \left(\sum_{j=1}^{k} i(j) \right) \\ &= \left(\sum_{j=1}^{k} j \right) \end{aligned}$$

Step 3: Solve (10) for $k \in \mathbb{N}$
Solving (10) for k to determine the conditions for termination gives:

$$\begin{aligned} & \exists k \in \mathbb{N} \left((s = 100) \left[s(k)/s \right] \right) \\ \equiv \;& \exists k \in \mathbb{N} \left(\sum_{j=1}^{k} j = 100 \right) \\ \equiv \;& \exists k \in \mathbb{N} \left(\frac{k(k+1)}{2} = 100 \right) \\ \equiv \;& \exists k \in \mathbb{N} \left(\frac{k^2 + k}{2} = 100 \right) \\ \equiv \;& \exists k \in \mathbb{N} \left(k^2 + k = 200 \right) \\ \equiv \;& false \end{aligned}$$

Step 4: Analyse the result

Because (10) can't be satisfied, we can reason that s will never equal 100 in any whole number of iterations. Therefore, under this initialisation, the guard will never be negated and the loop will never terminate. Solving the quadratic equation on the second last line of the above gives $k = 13.65$ (k can't be negative so we are only concerned with positive roots). We can reason from this, that up to the 13th iteration, the loop is making progress toward the termination state. That is $s < 100$ and s is increasing toward 100. However, the guard is not negated, and after iteration 14, the next whole number iteration, $s > 100$, and the loop will fail to progress. ■

The prototype tool automates steps 1 and 2 and the formulation of (10). As this tool is not a theorem prover, it is left to the human verifier to determine if (10) is satisfiable. A human verifier may decide, however, to discharge such a proof with the aid of a theorem prover. The output from the prototype tool for Program 2 is shown in Fig. 5.

When the difference equations for all loop guard variables are not solvable, it is possible to present the termination conditions as

$$\exists k \in \mathbb{N} \left(\neg G_k \right) \tag{11}$$

where $G_k \equiv G\left[v_k / v \right]$ for all variables $v \in \mathbf{V}(G)$. When read in conjunction with the loop progress conditions, PC, the iterative-form invariant, R_{k+1}, and the loop precondition, termination conditions presented in this form are useful for guiding reasoning about termination. For example, the difference equation solutions for x and y in the greatest common divisor example were not able to be solved by the prototype. In this case (11) was applied by the prototype, yielding the termination conditions and guidance for reasoning about them shown in Fig. 4.

5 Conclusions and Future Work

In this paper we have presented repeatable and automatable techniques for deriving loop progress and termination conditions. We have demonstrated the application of the techniques and discussed the use of their results in assisting verification, re-engineering and documentation of loops.

The techniques presented are applicable in all of the cases that the well-founded-sets methods of (Floyd 1967, Manna 1974) can be applied. The benefit of the approach presented in this paper is that loop progress and termination conditions can be expressed without needing to calculate an invariant. The methods presented here are not dependent on the provision of any formal documentation for the loop being analysed. As such, they are useful for the verification, documentation and re-engineering of loops in legacy programs for which existing formal methods are not applicable.

The approach presented forms a basis for investigating termination in both theorem proving based verification and human reasoning based code inspection processes. The progress conditions method and termination conditions methods have both been implemented in a prototype tool which has been discussed.

Most ongoing work in regard to investigating termination conditions involves the definition of further difference equation solution, and the extension of the logical and algebraic simplification capabilities of the prototype tool as well as an extension of the tool to handle real languages like Java. Work is planned that will focus on the integration of this tool with a theorem prover or equation solver to automate some of the reasoning steps, presently assigned to the human verifier.

A preliminary study with a small number of undergraduate students has found that they are capable of using the reports generated by the prototype to reason about loop progress and termination. A larger study, planned for completion in early 2004, will compare inspection yields and inspection rates from student inspection teams using the reports generated by the prototype to those of students inspecting with a traditional code inspection checklist. The study is being conducted using a module of search and partition algorithms with a variety of known, seeded defects.

We believe that the approaches presented in this paper and the corresponding tool provide support, not only for improving the repeatability of verification activities, but for encouraging a formal approach to investigating program correctness.

References

Alagic, S. & Arbib, M. (1978), *The Design of Well Strutcured and Correct Programs*, Springer-Verlag.

Back, R. (1988), 'A calculus of refinements for program derivation', *Acta Informatica* **25**, 593–625.

Backhouse, R. (1986), *Program Construction and Verification*, Prentice Hall.

Dijkstra, E. (1975), 'Guarded commands, nondeterminacy and formal derivation of programs', *Comm. ACM* **18**, 453–457.

Dijkstra, E. & Scholten, C. (1989), *Predicate Calculus and Program Semantics*, Springer-Verlag.

Dromey, R. (1989), *Program Derivation*, Addison-Wesley.

Dromey, R. & Pan, S. (1996), 'Re-engineering loops', *The Computer Journal* **39**(3).

Floyd, R. (1967), Assigning meanings to programs, *in* J. T. Schwartz, ed., 'Mathematical Aspects of Computer Science', American Mathematical Society, pp. 19–32.

Gannod, G. & Cheng, B. (1996), 'Strongest postcondition semantics as the formal basis for reverse enigineering', *Automated Software Engineering* **3**, 139–164.

Hoare, C. (1969), 'An axiomatic basis for computer programming', *Communications of the ACM* **12**(10), 576–583.

Manna, Z. (1974), *Mathematical Theory of Computation*, McGraw-Hill.

Pan, S. (1994), Software Quality Improvement, Specification, Derivation and Measurement using Formal Methods, PhD thesis, School of Computing and Information Technology, Griffith University.

Powell, D. (2002*a*), Deriving verification conditions and program assertions to support software inspection, *in* 'Proc. 9th Asia Pacific Software Engineering Conference'.

Powell, D. (2002*b*), Formal Methods For Verification Based Software Inspection, PhD thesis, Griffith University.