STEPWISE PROGRAM DERIVATION

R.G. Dromey and D. Billington
Programming Methodology Research Group
School of Computing and Information Technology
Griffith University, Nathan Qld 4111, Australia

ABSTRACT

Our understanding of the program derivation process has evolved to the point where it can be described in terms of a clearly defined sequence of steps. In this paper, we will identify those steps and show how they may be used to derive programs from formal specifications.

In describing the program derivation process we will focus on two things, its broad structure, and some detail for each of the principal steps. The suggestions made are practical and easily integrated with conventional methodologies for handling larger problems. The detailed steps described also provide the basis for the construction of a system for computer-assisted program derivation from formal specifications.

1. THE STRUCTURE OF THE PROGRAM DERIVATION PROCESS

The ideal we seek in program derivation is to begin with a formal specification of a problem, and then to proceed by a well-defined process of transformation and refinement until eventually a program implementation is derived. The process should be *constructive*, that is, at each stage in the development, the original, or derived specifications, should guide the process.

The process we will present here describes a sequence of heuristics which aspire to that ideal. It is best suited to deriving programs for precisely specified problems that require either single or nested loop structures. Such structures represent the building blocks for larger, more elaborate program structures. The task of handling these larger structures becomes one of identifying well-defined and precisely specified sub-problems, deriving their solutions, and composing them accordingly. We are fortunate that a large class of problems, but not all, submit easily to this kind of treatment.

The structure of the derivation process, at its broadest interpretation, involves essentially three distinct phases.

Phase I - Obtaining a Suitable Primary Specification

> Identification of a precondition/postcondition specification (Q,R) that is deemed to be suitable to guide the formal derivation process. A by-product of this phase is an appropriate initialization for the iterative structure.

Phase II - Obtaining Secondary Specifications

> Identification of ancillary specifications, and commands that may be used directly to guide the derivation of the body of the iterative structure. A by-product of this phase is the identification of the guard for the iterative structure.

Phase III - Derivation of the Body of the Iterative Structure

> Use of the results from the second phase to derive the body of the iterative structure. (In the course of this phase the need to derive additional iterative structures may be discovered - such structures must be derived accordingly.)

1.1 Obtaining a Suitable Primary Specification

Not all precondition/postcondition specifications that are adequate for defining a problem are suitable for guiding the derivation of a program. Two common problems with postconditions are

- they are not strong enough†
- they do not contain enough free variables.

Experienced programmers have no difficulty in identifying and rectifying either of these situations.

Our objective here is, not to rely on experience, but instead to provide an effective procedure for assessing the suitability of a postcondition specification.

We will consider postconditions consisting of sets of conjuncts. To assess the suitability of such a postcondition for derivation, it is convenient to classify its conjuncts according to their strength relative to the precondition Q after their free variables have been initialized.

Upon initialization each conjunct in the postcondition is either

(a) weaker than or the same strength as the precondition, or

(b) stronger than the precondition††

Using this classification scheme for postcondition conjuncts we can define two important relations, the base condition, and the termination condition.

Base Condition

> The base condition H is formed by the conjunction of those original components of the postcondition which, upon initialization of free variables, are at least as weak as the precondition.

Termination Condition

> The termination condition T, is formed by the conjunction of those original components of the postcondition which upon initialization of free variables are stronger than the precondition.

Using these definitions the postcondition R assumes the form

$$R \equiv H \wedge T$$

The initialization we seek is one which makes H as near in strength to R as possible, and T as weak and as simple as possible relative to R (note $R \Rightarrow H$ and $R \Rightarrow T$).

Frequently upon initialization components of R are reduced to TRUE or to the precondition itself - these components are directly assigned to the base condition. The following examples illustrate these ideas.

---

† In some cases, a specification may be weakened to allow the solution of a sub-problem.

†† A relation X is weaker than or the same strength as a relation Y if the implication $Y \Rightarrow X$ holds.

Example 1 - Conjunct Analysis

Q: $N \geq 1$
R: $(\forall j : 1 < j \leq i : a_{j-1} \leq a_j) \wedge i = N \wedge perm(a, A, N)$†

Initialization

$i, a := 1, A$

Conjunct	Initialized Form	Classification
1. $(\forall j : 1 < j \leq i : a_{j-1} \leq a_j)$	$(\forall j : 1 < j \leq 1 : A_{j-1} \leq A_j)$ $\equiv$ TRUE	H - Component
2. $i = N$	$1 = N$	T - Component
3. $perm(a, A, N)$	$perm(A, A, N)$ $\equiv$ TRUE	H - Component

It follows that

H : $(\forall j : 1 < j \leq i : a_{j-1} \leq a_j) \wedge perm(a, A, N)$
T : $i = N$

Example 2 - Conjunct Analysis

Q : $N \geq 0$
R: $a^2 \leq N < (a+1)^2$

Initialization

$a := 0$

Conjunct	Initialized Form	Classification
1. $a^2 \leq N$	$0^2 \leq N$ $\equiv 0 \leq N$	H - Component
2. $N < (a+1)^2$	$N < (0+1)^2$ $\equiv N < 1$	T - Component

It follows that

H: $a^2 \leq N$
T: $N < (a+1)^2$

The postcondition initialization and conjunct analysis identifies the

(a) components of the postcondition that are satisfied by the initialization - the H components

(b) components of the postcondition not guaranteed to be satisfied by the initialization - the T components.

---

† perm (a,A,N) - denotes the fact that the partitioned sequence $(a_1,a_2,...,a_N)$ is a permutation of the original sequence $(A_1,A_2,...,A_N)$

An additional requirement must be placed upon T, namely that, $\neg\Box T$ should serve as a guard for the iterative process that must establish R. It follows if $\neg\Box T$ is to serve as a guard in an implementation it should involve only a simple relation that does not contain any quantifiers or specifiers (i.e. $\forall$, $\exists$, $\sum$, Π, #). Whenever a termination condition that contains a quantifier or specifier is found, it signals the need to introduce a new free variable that will allow the quantified relation to be removed from T. The following example illustrates these points.

Example - Partitioning Specification

Stage (1) - unacceptable form for T

$$R : (\forall p : 0 < p \le i : a_p \le X) \wedge (\forall q : i+1 \le q < N+1 : a_q \ge X) \wedge perm(a,A,N) \wedge 0 \le i \le N$$

Initialization: i,a := 0,A
H: $(\forall p : 0 < p \le i : a_p \le X) \wedge perm(a,A,N) \wedge 0 \le i \le N$
T: $(\forall q : i+1 \le q < N+1 : a_q \ge X)$ - unacceptable

Stage (2) - acceptable form for T - free variable j introduced

$$R_D : (\forall p : 0 < p \le i : a_p \le X) \wedge (\forall q : j \le q < N+1 : a_q \ge X) \wedge perm(a,A,N) \wedge 0 \le i \le N \wedge j = i+1$$

Initialization: i,j,a := 0,N+1,A
H: $(\forall p : 0 < p \le i : a_p \le X) \wedge (\forall q : j \le q < N+1 : a_q \ge X) \wedge perm(a,A,N) \wedge 0 \le i \le N$
T: $j = i+1$ - acceptable

Deliverables - First Phase of Derivation

The overall outcome of the first phase of a derivation should be the following:

Initialization
- an initialization that establishes at least part of the postcondition R_D

R_D
- a constructive postcondition specification that is suitable for guiding the derivation of a program that will satisfy it. For such specifications $R_D \Rightarrow R$.

H
- the base condition, that is, the components of R_D that are satisfied by the initialization of free variables

T
- the termination condition, that is, the components of R_D that are not satisfied by the initialization of free variables

The base condition H and the termination condition T both play important parts in the next phase of the derivation.

1.2 Obtaining the Secondary Specifications

The other specifications needed to provide a framework for the constructive derivation of programs, are the invariant, and the variant function. In this section, we will describe procedures for deriving these specifications from the primary specification. We will also show how the guard for an iterative process and variant commands are obtained.

Invariant

The specification of central importance for proceeding with a derivation is the invariant P_D. An invariant is a relation that is used to model or characterize the dynamic behaviour of an iterative process. It provides a concise formal description of the properties that are established upon initialization, maintained with each iteration, and held to be true upon termination. An invariant is both a precondition and a postcondition of an iterative structure. Most importantly, it can be used in a weakest precondition calculation to derive the body of an iterative structure.

At this point, we are interested in two things

- the form that invariants should take, and
- how to obtain an invariant.

The usual advice given is that an invariant is obtained by weakening a postcondition. Here we wish to be much more prescriptive about how to obtain invariants and about what form they should take. The principle of derivation captures the appropriate advice.

Principle of Derivation

A knowledge of the base condition and the termination condition allows us to interpret the derivation process as one of keeping the base condition satisfied while changing the free variables in the termination condition to satisfy it as well. When both H *and* T *are satisfied, the postcondition is established.*

This interpretation of the derivation process immediately suggests how the invariant may be obtained and what form it may take. Two components are involved:

(i) Since the objective is to keep the base condition H satisfied, it should form one component of the invariant.

(ii) The other component of the invariant should be obtained by weakening the termination condition to obtain a relation that is satisfied, not only upon termination, but also upon initialization and after each iteration as well. The weakened termination condition that has the required properties is referred to as the cotermination condition C.

The invariant P is therefore of the form

$$P \equiv H \wedge C \qquad \text{where } T \Rightarrow C$$

The values of free variables used to initialize the iterative process, together with the termination condition, usually suggest bounds for free variables in C. To illustrate this process we may continue with our earlier example.

Example - Partitioning Problem

Using the earlier results, we have

Initialization: i,j,a := 0,N+1,A
T: j = i+1

Weakening T and taking into account the bounds on free variables indicated by the initialization (i.e., i=0 and j=N+1), we get the cotermination condition

C: $0 \le i < j \le N+1$ - cotermination condition

Combining the cotermination condition C with the base condition H yields the invariant $\Box$P.

P : $(\forall p : 0 < p \le i : a_p \le X) \wedge (\forall q : j \le q < N+1 : a_q \ge X) \wedge perm(a,A,N) \wedge 0 \le i < j \le N+$

Using the scheme we have outlined for obtaining an invariant ensures that the postcondition ends up as one of the disjuncts of the invariant - this is the most desirable relationship between these two specifications.

To emphasize this relationship between invariant and postcondition, consider a second example.

Example - Exponentiation Problem

Q : $X > 0 \wedge Y \ge 0$ - precondition
R : $z = X^Y$ - postcondition
R_D : $z = X^y \wedge y = Y$ - transformed postcondition

Initialization: y,z := 0,1
H: $z = X^y$ - base condition
T: $y = Y$ - termination condition

C: $0 \le y \le Y$ - cotermination condition

P_D: $z = X^y \wedge 0 \le y \le Y$ - invariant (H $\wedge$ C)

It can be seen that the invariant P_D includes the postcondition R_D as a disjunct by rewriting the former in the equivalent form

P_D: $z = X^y \wedge (y = 0 \vee y = 1 \vee \ldots \vee y = Y)$

which expands to

P_D : $(z = X^y \wedge y = 0) \vee (z = X^y \wedge y = 1) \vee \ldots \vee (z = X^y \wedge y = Y)$
$\leftarrow \quad R_D \quad \rightarrow$

Variant Function

The other important consideration in the second phase of the derivation process is the identification of the variant function and the related variant command(s) that ensures the variant function is decreased with each iteration.

Variant functions are used to characterize the termination and progression properties of iterative mechanisms. A variant function may be related to the corresponding termination condition for an iterative process.

We have just seen how the termination condition was weakened to obtain the cotermination condition. To identify the variant function, we again want to find a condition that holds upon initialization, after each iteration, and upon termination. The difference here is that we must satisfy the requirement that the variant function is bounded below by zero. This can be done by carrying out the following steps.

(i) First we rewrite the termination condition so that it involves an equality relation (this may involve the addition of an unspecified constant to a term).

(ii) Next we substitute initialization values for the free variables in the termination condition.

(iii) Then information from the precondition is used to substitute for any constants† appearing in the termination condition. Constants should be replaced by their lower bound value plus any indicative range.

(iv) The weakened relation resulting from step (iii), when simplified, should reduce to a standard form which indicates directly a new relation called a variant condition. The variant condition possesses a term which is reducible with each iteration, and is greater than or equal to zero upon initialization, after each iteration, and upon termination. The set of standard forms may be used to create a set of rules for inferring the corresponding variant conditions.

(v) The term in the variant condition that remains greater than or equal to zero is the variant function, t.

Details of the five primary inference rules involving variables v and w, the constants C, F and G and lower bounded ranges X^+ and Y^+ are as follows.†

Variant Condition Inference Rules††

(1)	(v,X)	$\cdot=\cdot$ (w,Y^+)	$\rightarrow$	$w - v \geq 0$
(2)	(v,X^+)	$\cdot=\cdot$ (w,Y^+)	$\rightarrow$	$v + w \geq 0$
(3)	(v,X)	$\cdot=\cdot$ (C,Y^+)	$\rightarrow$	$C - v \geq 0$
(4)	(v,X^+)	$\cdot=\cdot$ (C,Y^+)	$\rightarrow$	$v - C \geq 0$
(5)	(v,X^+)	$\cdot=\cdot$ (C,Y)	$\rightarrow$	$v - C \geq 0$

Two auxiliary rules that are often useful in reducing relations to one of the standard forms are:

(6) $(F, (X + Y)^+) = (G, X^+) + (H, Y^+)$

---

† By constants in this context we mean free variables which are fixed with respect to any given computation.

† The notation X^+, indicates a lowest possible value of X with other possible values of X+1, X+2, ...

†† The symbol "$\cdot=\cdot$" is used to indicate that the equality relation is not necessarily preserved.

(7) $(F, 0^+) = (G, X^+) - (H, Y^+)$ where $X \geq Y$

Often in practice it is not necessary to go through such an elaborate process to identify the variant function from the termination condition. However, if we are interested in mechanizing variant function identification such a process is appropriate. Two examples that illustrate the process are as follows.

When substitutions are made in the termination condition we use an ordered pair notation of the form (variable/constant, value/range). For example, when we substitute 0 for the variable i we use (i,0).

Example - Partitioning Problem

Continuing with our partitioning problem introduced earlier, we have

Q:	$N \geq 0 \rightarrow (N, 0^+)$	- precondition
T:	$j = i + 1$	- termination condition
Init :	$i := 0; j := N + 1 \rightarrow (i,0), (j, N + 1)$	- initialization

Variant function identification

$j = i + 1$	$\rightarrow (j, N + 1) = (i + 1, 0 + 1)$	- variable substitution
	$\rightarrow (j, N + 1) = (i + 1, 1)$	- simplification
	$\rightarrow (j, 0^+ + 1) .=. (i + 1, 1)$	- constant substitution
	$\rightarrow (j, 1^+) .=. (i + 1, 1)$	
	$\rightarrow j - i - 1 \geq 0$	- Rule (1)
	$\rightarrow j - i - 1$	- variable function

Example - Quotient Remainder Problem

An example where it is necessary to introduce a constant in the process of deriving the variant function is as follows:

R:	$X = q * D + r \wedge 0 \leq r \leq D$
Q:	$X \geq 0 \wedge D > 0 \rightarrow (X, 0^+), (D, 1^+)$
T:	$r < D$
Init:	$r := X \rightarrow (r, X)$

Variant function identification

$r < D$		- termination condition
$D = r + F$	$\rightarrow D = (r, X) + F$	- variable substitution
	$\rightarrow (D, 1^+) .=. (r, 0^+) + (F, 0^+)$	- constant substitution
	$\rightarrow (r, 0^+) .=. (D, 1^+) - (F, 0^+)$	
	$\rightarrow (r, 0^+) .=. (D - F, 0^+)$	
	$\rightarrow r - D + F \geq 0$	- Rule (4)
	$\rightarrow r - D + F$	- variant function

Variant Commands

Once a variant function has been identified, it is then usually a simple matter to identify the command or commands that will decrease it. Such commands are referred to as variant commands. Variant commands play a vital constructive role - they are used in calculations with an invariant to derive the body of an iterative structure. The form that variant commands take follows directly from the form of the variant function. Examples serve as the best way of characterizing the process of obtaining variant commands.

Example - Partitioning Problem

Earlier with our partition problem, we found that its variant function was

```
t: j - i - 1
```

There are two ways of decreasing this variant function, either by increasing i, or decreasing j. Obvious variant commands are

```
VC1 :   i := i + 1
VC2 :   j := j - 1
```

In general, there should be a corresponding variant command for each free variable in the variant function. As a base strategy, we choose to construct variant commands that will decrease the variant function by one when executed. Sometimes, however, depending on the problem and in the interests of efficiency, we may choose variant commands that will decrease a variant function by larger amounts. This occurs particularly with problems that have strong preconditions which permit the application of divide-and-conquer strategies. Whenever a variant command is chosen that decreases a variant function by more than one, it is particularly important to check the guard to ensure termination, and that the variant function remains bounded below by zero.

Example - Quotient Remainder Problem

The variant function derived earlier for this problem is

```
t :      r - D + F
```

Possible variant commands are

```
VC1 :   r := r - 1
VC2 :   r := r - D
```

The choice of the variant command "r := r - D" is influenced by the knowledge that the problem involves a divisor D.

Guard

The termination condition T, identified in the first phase of the derivation, may be used to derive a suitable guard B for the corresponding iterative process. Clearly, while the termination condition is not satisfied, it is necessary to continue to decrease the variant function.

The weakest, and therefore the preferred guard that can be derived from the specification, is the one obtained by simply negating the termination condition, i.e.

$B \equiv \neg T$ - weakest guard

The weakest guard corresponds to the strongest termination condition.

Example - Partitioning Problem (see earlier)

T:	$j = i+1$	- termination condition
B:	$j \neq i+1$	(weakest guard, $\neg$ T)

Example - Quotient Remainder Problem (see earlier)

T:	$r < D$	- termination condition
B:	$r \geq D$	- guard

Deliverables - Second Phase of Derivation

The second phase of development should yield the following constructs:

P_D - an invariant specification that can be used to guide the derivation of the iterative body

t - a variant function that must be decreased with each iteration

VC - one or more variant commands that decrease the variant function

B - a guard for the iterative construct

1.3 Derivation of the Iterative Structure Body

How the derivation of the body proceeds depends on two things

- the form of the invariant, and
- the form of the variant function, and its associated variant commands.

An inspection of the invariant that must be maintained as the variant function is decreased, reveals whether

- the invariant contains any of the specifiers, $\sum, \prod, \#$

  e.g. $s = \sum(A_j : 0 < j \leq i) \wedge 0 \leq i \leq N$

- the invariant contains only variant function free variables

  e.g. P: $(\forall j: 0 < j \leq i : A_j \neq X) \wedge (n = N \text{ cor } A_{i+1} = X) \wedge 0 \leq i \leq n \leq N$

  t: $n-i$

  free variables: i,n

- the invariant contains a mix of variant function free variables and other free variables

  e.g. P: $(\forall j: 1 \leq j \leq i : m \geq A_j) \wedge 1 \leq p \leq i \wedge m = A_p \wedge 1 \leq i \leq N$

  t: $N-i$

Variant free variables: i

Other free variables: m,p

- the invariant contains nested quantifiers or specifiers

e.g. P: $(\forall k : 1 \le k < i : (\forall p: 1 \le p \le k : (\forall q : k+1 \le q \le N : a_p \le a_q)))$
$\wedge\ 1 \le i \le N \wedge$ perm(a,A,N)

t: N-i

Each of these cases requires a different treatment if the iterative structure body is to be successfully derived. These cases will be considered in detail in the next section. What we intend to do in this section is sketch the overall process.

At the time when we are in a position to derive the body for an iterative structure, we already have in hand, an initialization, a guard, an invariant, and one or more variant commands. Schematically we have

GIVEN:

⌈Initialization⌉

do ⌈GUARD⌉ →

{INVARIANT holds}

⌈Variant⌉Command(s)⌈

{Require Invariant to hold}

od

Unfortunately, the invariant alone does not represent a strong enough condition to allow a variant command to execute and maintain the invariant. Instead, a stronger condition called the weakest precondition must hold in order to execute a variant command and re-establish the invariant. The weakest precondition includes a specification for the command that must go in the loop body to maintain the invariant. Schematically

REQUIRE

⌈Initialization⌉

do ⌈GUARD⌉ →

{WEAKEST PRECONDITION must hold}†

⌈Variant⌉Command(s)⌈

{INVARIANT re-established}

od

† To be strictly correct, the weakest precondition or a stronger condition must hold.

At this point it is useful to introduce the definition of the weakest precondition for an assignment statement x := e, i.e.

$$wp("x := e", R) \equiv R_e^x \qquad - \quad (1)$$

R_e^x is the predicate R with all free occurrences of x simultaneously replaced by an executable expression e. What this definition expresses is that R_e^x must hold prior to the execution of "x := e" for this command to execute and establish R.

Usually, when a weakest precondition calculation is performed with an invariant, the resulting formula will include the invariant. An inspection is then needed to reveal the required commands for the loop body. However, these commands can be constructed from the results of routine calculations. These calculations involve the weakest precondition and a new function called the guard generator.

The guard generator takes an assignment command, S, and an invariant, P, and returns a formula, gg(S,P), such that

(i) $P \wedge gg(S,P) \Rightarrow wp(S,P)$, and

(ii) gg(S,P) does not duplicate any of the constraints on variables already imposed by P.

Provided that P is in an appropriate form, (see Billington and Dromey 1991), the guard generated from S and P, gg(S,P), is defined below.

(1) If P is a simple relation, or the negation of a simple relation, then

$$\text{then } gg(S,P) = \begin{cases} TRUE & \text{if } P \Rightarrow wp(S,P) \\ wp(S,P) & \text{otherwise} \end{cases}$$

(2) If $P = (\forall j : L : G(j))$ and $wp(S,P) = (\forall j : L' : G'(j))$ then
$gg(S,P) = (\forall j : (L' - L) : G'(j)) \wedge (\forall j : (L \cap L') : gg(S,G(j)))$.

(3) If $P = (\exists j : L : G(j))$ and $wp(S,P) = (\exists j : L' : G'(j))$ then
$gg(S,P) = (\forall j : (L - L') : \neg G(j)) \wedge (\forall j : (L \cap L') : gg(S,G(j)))$.

(4) If $P = G \wedge H$ or $P = G \vee H$ then
$gg(S,P) = gg(S,G) \wedge gg(S,H)$.

Weakest Precondition Calculations

Returning to the original task in this phase of the derivation, we must perform weakest precondition and/or guard generator calculations that will identify:

(a) Suitable guards for variant commands, e.g.

$P: (\forall j : 1 \le j \le i : m \ge A_j) \wedge m \in A[1 .. i] \wedge 1 \le i \le N$

t: N-i

VC: i := i+1

$wp("i := i+1", P) \equiv (\forall j : 1 \le j \le i+1 : m \ge A_j) \wedge m \in A[1 .. i+1] \wedge 1 \le i+1 \le N$

$gg("i := i+1",P) \equiv i \neq N \wedge m \geq A_{i+1}$

GC: if $m \geq A_{i+1} \rightarrow i := i+1$ fi - guarded command

(b) Assignment statements that must be used in concert with variant commands in order to maintain the invariant for the iterative structure body. In some cases these assignments must be guarded. An example showing how an unknown assignment "s := exp" is derived is:

P: $s = \sum(A_j : 0 < j \leq i) \wedge 0 \leq i \leq N$
t: $N - i$
VC: $i := i+1$

$wp("i,s := i+1,exp",P)$
$\equiv exp = \sum(A_j : 0 < j \leq i+1) \wedge 0 \leq i+1 \leq N$
$\equiv exp = \sum(A_j : 0 < j \leq i) + A_{i+1} \wedge 0 \leq i+1 \leq N$
$\equiv exp = s+A_{i+1} \wedge 0 \leq i+1 \leq N$ - using P

Command: $i,s := i+1, s+A_{i+1}$

(c) A complex guard that requires the separate derivation of an iterative structure, e.g.

P: $(\forall k : 1 \leq k < i : (\forall p : 1 \leq p \leq k : (\forall q : k+1 \leq q \leq N : a_p \leq a_q))) \wedge 1 \leq i \leq N$
t: $N - i$
VC: $i := i+1$

$wp("i := i+1",P) \equiv$
$(\forall k : 1 \leq k < i+1 : (\forall p : 1 \leq p \leq k : (\forall q : k+1 \leq q \leq N : a_p \leq a_q))) \wedge 1 \leq i+1 \leq N$

$gg("i := i+1",P) \equiv i \neq N \wedge (\forall q : i+1 \leq q \leq N : a_i \leq a_q)$

Assignment Construction Inference Rules

When the technique of using unknown expressions is used with weakest precondition calculations (as in (b) above), it is important to define a set of accompanying construction rules that allow the identification of assignment commands. The purpose of these rules is to identify equalities that ensure constraints not implied by the conjunction of the invariant and the guard are satisfied. The equalities then provide a basis for identifying corresponding assignments. For example, if a constraint $a = b+1$ in a weakest precondition calculation is not implied by the invariant P and the guard B, then the assignment $a := b+1$ will ensure the constraint is satisfied.

The constructive rules we will consider fall into two categories:

(i) Generated guard
Constructive Equalities

(ii) Generated guard $\wedge$ Precondition
Constructive Equalities

In each of these rule types, the generated guards involve unknown expressions (e.g. exp, exp_k, and exp_l). Sample rules are:

(a) $\dfrac{exp \geq A}{exp = A}$

(b) $\dfrac{A\,exp = A\,x}{exp = x}$

(c) $$\frac{A_i < A_j \wedge exp_k < A_j \wedge perm(Exp, A)^{\dagger}}{exp_k = A_i}$$

(d) $$\frac{A_i > A_j \wedge exp_k \leq exp_l \wedge perm(Exp, A)^{\dagger}}{exp_k = A_j \wedge exp_l = A_i}$$

To enhance the power of program derivation methods, an extensive search is needed to find assignment construction rules. We will see later in the next section, how the first two of the rules above are used in a derivation.

This completes our broad background of the major phases in program derivation.

† In these examples A and exp represent sequences. A_i and A_j are elements of the sequence A and exp_k and exp_l are elements of the sequence Exp.

2. THE STEPS IN PROGRAM DERIVATION

A simple process-oriented view of program derivation will now be presented. Viewed as a process, ten major steps are needed to derive a program from a formal specification. (See Fig. 1.) We will now consider each of the steps in detail, and in the process work through a complete example.

1. Problem Specification
2. Postcondition Initialization
3. Specification Suitability Test
4. Specification Modification (if necessary)
5. Invariant Identification
6. Variant Function Identification
7. Guard Identification
8. Summarize Progress
9. Derivation of Program Body (one or more refinements)
10. Program Body Finalization

Figure 1. The Steps in Program Derivation

STEP 1 PROBLEM SPECIFICATION

Purpose:

To obtain a formal precondition/postcondition specification that provides a precise and unambiguous statement of the problem to be solved.

Process:

1(a) Formulate a precondition that expresses the constraints on the input data. Include ranges for free variables where possible.

1(b) Formulate a postcondition that expresses the constraints on the output data. Include ranges of free variables where possible.

Example - (Step 1) - Finding the Maximum and its position†

Q: $N \geq 1$ - precondition

R: $(\forall j : 1 \leq j \leq N : m \geq A_j) \wedge m = A_p \wedge 1 \leq p \leq N$ - postcondition

The postcondition expresses the fact that m is equal to the element at position p, and m is greater than or equal to the N elements in the sequence $(A_1, A_2, ..., A_N)$.

† We have chosen this example so that a comparison with Dijkstra's original derivation may be made (See Dijkstra (1976)).

Remarks:

- Uppercase is used for precondition free variables.
- Lowercase is used for free variables in the postcondition assigned by the program.
- The first order predicate calculus, together with notations for sums, products, counts and sets is used to express problems.

STEP 2: POSTCONDITION INITIALIZATION

Purpose: *To find an initialization of free variables in the postcondition that will lead to the identification of base condition, H, and termination condition, T, components of that specification.*

Process:

2(a) Examine the postcondition, identify all free variables, then divide them into two classes - those associated with ranges, and other free variables.

2(b) **IF** the postcondition contains no range free variables

THEN it should be modified according to STEP 4 (see below).

ELSE IF

the postcondition contains only range free variables

THEN

compute the weakest precondition that the initialization of those free variables will establish the postcondition and then perform a conjunct analysis with respect to the precondition. Those components of the postcondition reduced to TRUE, to the precondition, or to a relation weaker than or equal to Q by this calculation belong to the base condition, the remaining components belong to the termination condition.

ELSE IF

the postcondition contains a mix of range free variables and other free variables

THEN

(i) initialize the range free variables to associated bound values

(ii) assign other free variables to distinct unknown expressions

(iii) perform a weakest precondition calculation for the initialization multiple assignment with the postcondition

(iv) identify any unknown using the precondition, the postcondition, and assignment construction rules where necessary.

2(c) Once the complete initialization has been identified, perform the weakest precondition calculation with the complete initialization multiple assignment.

2(d) Identify the base condition and the termination condition using the approach in step 2(b).

Example - (Step 2) - Finding the Maximum and its Position

Initialization: $p,m := 1,A_1$

$wp("INIT",R) \equiv (\forall j : 1 \le j \le N : A_1 \ge A_j) \wedge A_1 = A_1 \wedge 1 \le 1 \le N$

H: $m = A_p \wedge 1 \le p \le N$ - base condition
T: $(\forall j : 1 \le j \le N : m \ge A_j)$ - termination condition

Remarks:

- The derivation of a program translates into a task of keeping the base condition satisfied while systematically changing the values of the variable(s) in the termination condition (and others where necessary) until it is satisfied as well.
- Bounds on the ranges of free variables are usually suitable candidates as initialization values.

STEP 3 SPECIFICATION SUITABILITY TEST

Purpose:

To establish whether the current form of the postcondition is suitable to derive a program from.

Process:

3. **IF** the termination condition identified in STEP 2 involves only a simple (preferably single) relation not containing any quantifiers or specifiers

   THEN

   proceed with the derivation (STEP 5)

   ELSE

   modify the specification (STEP 4).

Example - (Step 3) - Finding the Maximum and its Position

T: $(\forall j : 1 \le j \le N : m \ge A_j)$

The termination condition contains a quantifier which suggests that it should be modified before proceeding with the derivation - go to Step 4.

Remarks:

- A termination condition that involves a quantifier (e.g. $\forall$, $\exists$) or specifier (e.g. $\sum$, $\prod$, #, etc) is unlikely to yield a guard that is practical to implement in a programming language. Such complex relations can usually be transferred from the termination condition to the base condition by introducing extra free variables into the offending conditions.

- The most common problem with specifications is that they are not strong enough or they contain too few free variables.

STEP 4: SPECIFICATION MODIFICATION

Purpose:

To transform the postcondition specification so as to obtain a stronger or an equivalent specification that will pass the specification suitability test (STEP 3).

Process:

4(a) **IF** a quantified or specified expression was left in the termination condition by the previous initialization

THEN

replace a literal, a constant, a variable, or an expression by a new free variable and record the identity by adding the corresponding conjunct.

ELSE IF

the problem involves an equality relation in which there are no free variables on one side of the relation

THEN

introduce a free variable to that side of the relation.

ELSE IF

the problem contains a search of some kind

THEN

ensure there is a predicate that describes where the value or condition sought is not present.

ELSE IF

none of the above situations apply

THEN

modify the specification by strengthening it (i.e., by adding conjuncts), or by an equivalence transformation, or by weakening the specification to solve a sub-problem first.

4(b) After transforming the postcondition, repeat steps 2 and 3.

Example - (Step 4) - Finding the Maximum

$R: (\forall j : 1 \le j \le N : m \ge A_j) \wedge m = A_p \wedge 1 \le p \le N$

Introducing a new free variable i in place of N yields

$R_D: (\forall j : 1 \le j \le i : m \ge A_j) \wedge m = A_p \wedge 1 \le p \le i \wedge i = N$

Step 2 - Repeated

Initialization : $i,p,m := 1,1,A_1$ - INIT

$wp("INIT",R_D) \equiv (\forall j : 1 \le j \le 1 : A_1 \ge A_j)$
$\wedge A_1 = A_1 \wedge 1 \le 1 \le 1 \wedge 1 = N$
$H: (\forall j : 1 \le j \le i : m \ge A_j) \wedge m = A_p \wedge 1 \le p \le i$ - base condition
$T: i = N$ - termination condition

Step 3 - Repeated

$T: i = N$

The termination condition contains only a simple relation so we can proceed to STEP 5.

Remarks:

- A specification can be modified by making it stronger, weaker, or by an equivalence transformation.
- In some problems, it is important to combine the precondition with the postcondition to obtain a strong enough postcondition to permit a derivation.
- Weakening a postcondition to obtain a sub-problem should only be considered when all other alternatives have been exhausted.

STEP 5: INVARIANT IDENTIFICATION

Purpose:

To obtain an invariant from the base condition and termination condition

Process:

5(a) Weaken the termination condition to obtain a relation that is satisfied, not only upon termination, but also after each iteration and upon initialization.

Carry out this step by assigning bounded ranges to free variables in the termination condition. These ranges should be consistent with the initialization (STEP 2), the precondition, and the postcondition.

5(b) The formula resulting from the conjunction of the termination condition with the ranges derived in 5(a) should be examined and any conjuncts that do not necessarily satisfy the requirement of being satisfied upon initialization, after each iteration and upon termination should be deleted. The resulting formula is the cotermination condition.

5(c) Form the invariant by taking the conjunction of the base condition with the cotermination condition.

Example - (Step 5) - Finding the Maximum and its Position

$H: (\forall j : 1 \le j \le i : m \ge A_j) \wedge m = A_p \wedge 1 \le p \le i$
$T: i = N$

$C: 1 \le i \le N$ - cotermination condition
$P_D: (\forall j : 1 \le j \le i : m \ge A_j) \wedge m = A_p \wedge 1 \le p \le i \wedge 1 \le i \le N$ - invariant

STEP 6: VARIANT FUNCTION IDENTIFICATION

Purpose:

To use the termination condition, the precondition, and the initialization to construct a variant function and an accompanying set of commands that will decrease that function.

Process:

6(a) Rewrite the termination condition so that it involves an equality relation. Add an unspecified constant to achieve this if necessary.

6(b) Substitute initialization values for the free variables in the termination condition.

6(c) Use precondition relations to replace constants in the termination condition by lower bounded ranges. Reduce the resulting relation to a standard form.

6(d) Use the variant condition inference rules to infer the variant condition and variant function.

6(e) For each free variable in the variant function identify a command that will decrease the variant function. Such commands are called variant commands. Unless special knowledge of the context dictates otherwise, choose commands that decrease the variant function by one.

Example - (Step 6) - Finding the Maximum and its Position

Q:	$N \ge 1 \rightarrow$	$(N, 1^+)$	- precondition
T:	$i = N$		- termination condition
Init:	$i := 1 \rightarrow$	$(i, 1)$	- initialization (partial)

Variant function identification

$i = N \rightarrow$	$(i,1) = N$	- Variable substitution
$\rightarrow$	$(i,1) \cdot=\cdot (N, 1^+)$	- Constant substitution
$\rightarrow$	$N - i \ge 0$	- Rule (3)
$\rightarrow$	$N - i$	- Variant function
$\rightarrow$	$i := i + 1$	- Variant command

Remarks:

- Conversion to an equality relation is essential to ensure we end up with a variant function that is bounded below by zero.
- The substitution of ranges for constants effectively weakens the termination condition to yield an invariant relation that is greater than or equal to zero.
- Identification of the variant function is of central importance because it provides the clues for the next stages in the derivation.
- Variant commands play a key role in helping construct other commands by weakest precondition calculation.
- The underlying idea in identifying the variant function is to look for the "slope" of the termination condition at initialization.

STEP 7 GUARD IDENTIFICATION

Purpose:

To use the termination condition to derive a suitable guard for the iterative process.

Process:

7. Negate the termination condition to obtain the guard for the iterative process.

Example - (Step 7) - Finding the Maximum and its Position

T: $i = N$ - termination condition
B: $i \neq N$ - guard

Remarks:

- Negating the termination condition yields the weakest useful guard for an iterative process.
- While the guard is satisfied, it is necessary to decrease the variant function.

STEP 8 SUMMARIZE PROGRESS

Process:

8. Fill in details of the iterative structure, that is, the initialization, the guard, the invariant, the variant function, and the variant commands needed to derive the body.

Example - (Step 8) - Finding the Maximum and its Position

```
i,p,m := 1,1,A1;
do i ≠ N →
      "Decrease t while maintaining PD"
od

PD: (∀j : 1 ≤ j ≤ i : m ≥ Aj) ∧ m = Ap ∧ 1 ≤ p ≤ i ∧ 1 ≤ i ≤ N

t:   N - i
VC:  i := i+1
```

Remarks:

- It is important to summarize the results of the first seven steps before attempting to derive the body of the iterative structure.

STEP 9 DERIVATION OF PROGRAM BODY

Purpose:

To derive commands that change the free variables in the invariant so that it is always maintained as progress is made towards termination with each iteration.

Process:

9. **IF** (CASE I)

   a free variable is involved in a top-level equality relationship with $\sum$ or $\prod$

   THEN

   (i) Perform a multiple assignment weakest precondition calculation involving a variant command, together with the "equality" free variable† assigned to an unknown expression.

   (ii) Use substitution from the invariant, assignment construction rules, and simplification, to identify the unknown expression.

   ELSE IF (CASE II)

   a free variable is involved in a top-level equality relationship with the counting specifier #

   THEN

   (i) Perform a multiple assignment weakest precondition calculation involving a variant command, and the "counting" free variable.††

---

† By an equality free variable we mean a free variable involved in an equality relation, i.e. v = ...

†† By the "counting free variable" we mean a variable c in the context c = #(...)

(ii) Identify the guard for the multiple assignment. Repeat (i) and (ii) for any other variant commands.

(iii) Negate the disjunction of the guard(s) from the previous step to establish the guarded command that must execute when the "counting condition" does not hold.

ELSE IF (CASE III)

the invariant contains nested quantifiers or specifiers

THEN

(i) Perform a weakest precondition calculation with the variant command.

(ii) Identify the sub-goal that must be established.

(iii) Derive the program for the sub-goal using the same overall ten-step process.

ELSE (CASE IV)

(i) For each variant command, compute the weakest precondition/generated guard that it will execute and maintain the invariant, and thereby identify the corresponding guarded command.

(ii) Negate the disjunction of the guards from Step (i) to obtain a new guard†††

(iii) Perform a multiple assignment weakest precondition calculation involving all variant commands with any other free variables appearing in the resultant guard assigned to unknown expressions.

(iv) Use simplification rules, the invariant, and assignment construction rules to identify any unknown expressions to complete the guarded command structure.

(v) Ensure the invariant is maintained and the variant function decreased by commands identified in (iv).

---

††† To complete the derivation for some problems, it is necessary to use stronger guards than those derived by this strategy.

Example - (Step 9) - Finding the Maximum and its Position

For our example, CASE IV applies, i.e.

STEP (i)

$P: (\forall j : 1 \le j \le i : m \ge A_j) \wedge m = A_p \wedge 1 \le p \le i \wedge 1 \le i \le N$
$VC: i := i+1$

$wp("i := i+1",P) \equiv (\forall j : 1 \le j \le i+1 : m \ge A_j) \wedge m = A_p \wedge 1 \le p \le i+1 \wedge 1 \le i+1 \le N$
$gg("i := i+1",P) \equiv m \ge A_{i+1} \wedge i \ne N$

Therefore, $i \ne N$ is the guard and the guarded command is

$if\ m \ge A_{i+1} \rightarrow i := i+1$

STEP (ii)

Negating the guard we get

$m < A_{i+1}$

STEP (iii)

Performing the weakest precondition calculation we get

$wp("i,p,m := i+1,Exp1,Exp2",P)$
$\equiv (\forall j : 1 \le j \le i+1 : Exp2 \ge A_j) \wedge Exp2 = A_{Exp1} \wedge 1 \le Exp1 \le i+1 \wedge 1 \le i+1 \le N$
$\equiv (\forall j : 1 \le j \le i : Exp2 \ge A_j) \wedge Exp2 \ge A_{i+1} \wedge Exp2 = A_{Exp1} \wedge 1 \le Exp1 \le i+1 \wedge 1 \le i+1 \le N$

Considering conjuncts with unknowns one at a time

$Exp2 \ge A_{i+1} \rightarrow Exp2 = A_{i+1}$ - Assignment Construction

$Exp2 = A_{Exp1} \rightarrow A_{i+1} = A_{Exp1} \rightarrow Exp1 = i+1$ - Assignment Construction

Therefore the guarded command is

$m < A_{i+1} \rightarrow i,p,m := i+1, i+1, A_{i+1}$

Remarks:

- STEP 9 is clearly the most complex step in the derivation. There are some instances where it is more fruitful to focus first on changing a single variant function free variable, and then on negating the associated guard rather than pursue calculations with other variant commands independently.
- Care should be taken when negating the disjunction of a set of guards to ensure that the strongest valid guards are obtained.
- After negating the disjunction of the guards, it is important that all free variables referenced in the resulting guard are considered for change in the constructed multiple assignment.

- The two assignment construction rules used in our example are

  (i) $\dfrac{Exp \geq X}{Exp = X}$ - relational equality rule

  (ii) $\dfrac{A_{Exp} = A_x}{Exp = X}$ - index equality rule

  There are a number of other such assignment inference rules.

- Step (iv) of CASE IV can sometimes demand a sophisticated response that is beyond what is reasonable to mechanize, given our current understanding of the program derivation process. Our implementation of the step should therefore be cautious and conservative.

STEP 10 PROGRAM BODY FINALIZATION

Purpose:

To complete the textual representation of the derived program.

Process:

10. Fill in details of the program body including all guarded commands and any sub-goal components.

Example - (Step 10) - Finding the Maximum and its Position

```
i,p,m := 1,1,A1;
do i ≠ N →
    if m ≥ Ai+1 → i := i+1
    [] m < Ai+1 → i,p,m := i+1, i+1, Ai+1
    f i
od
```

CONCLUSION

In this paper we have interpreted the program derivation process as a sequence of well-defined steps and shown their dependencies.

Our objective has been to create a model of program derivation that is easier to teach and more amenable to implementation as a system for computer-assisted program derivation. The stepwise formulation provided satisfies both these requirements.

In the course of our investigation, we have gained a number of insights into the program derivation process. New suggestions have been made for assessing the suitability of a given specification for derivation. Base condition and termination condition specifications have been defined and shown to play a useful role in the derivation process. A number of rules and procedures for obtaining, guards, invariants, and variant functions have been provided. The problem of deriving the body of an iterative structure has also been considered and suggestions have been made that relate directly to the form of the invariant used in the derivation.

The method described has been used to derive programs for a significant number of formally specified small problems.

It has also provided the framework for the construction of a system for computer-assisted program derivation. This system which is still under development can already derive a substantial number and variety of programs that involve the processing of sequences.

BIBLIOGRAPHY

Backhouse, R.C., "Program Construction and Verification", Prentice Hall, London (1986).

Billington, D., and Dromey, R.G., "The Guard Generator: an aid in Deriving Loop Bodies". IEEE Trans. on Soft. Eng. (Submitted for Publication).

Dijkstra, E.W., "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J. (1976).

Dijkstra, E.W., and W.H.J. Feijen, "A Method of Programming", Addison-Wesley, Wokingham, England (1988).

Dijkstra, E.W. (Ed.), "Formal Development of Programs and Proofs", Addison-Wesley, Reading, Mass. (1990).

Dromey, R.G., "Program Derivation", Addison-Wesley, Sydney (1989).

Dromey, R.G., "Systematic Program Development", IEEE Trans. on Soft. Eng 24(1), 12-29 (1988).

Gries, D., "The Science of Programming", Springer-Verlag, New York, (1981).

Gries, D., "A Note on a Standard Strategy for Developing Loop Invariants and Loops", Science of Computer Programming 2, 207-214 (1982).

Hehner, E.C.R., "The Logic of Programming", Prentice-Hall, Englewood Cliffs, N.J. (1984).

Morgan, C. "Programming from Specifications", Prentice-Hall, London (1990).

Turski, W.M., "Computer Programming Methodology", Heydon, London (1978).

Turski, W.M., "On Programming by Iterations", IEEE Trans. on Soft. Eng., SE-10, 175-8 (1984).

Wirth, N. "Program Development by Stepwise Refinement", Comm. ACM. 14, 221-7 (1971).