A discipline for program development

R. Geoff Dromey
University of Wollongong

A DISCIPLINE FOR PROGRAM DEVELOPMENT

R. Geoff Dromey

Department of Computing Science
University of Wollongong
P.O. Box 1144, Wollongong NSW 2500
AUSTRALIA

ABSTRACT

A constructive method of program development is presented. It is based on a simple strategy for problem decomposition that is claimed to be more supportive of goal-oriented programming than the Wirth-Dijkstra top-down refinement method. The strategy can minimize case analysis, simplify constructive program proofs, and ensure a correspondence between program structure and data structure.

1. INTRODUCTION

The semantics of a program may be conveniently interpreted in terms of a mapping between sets of initial states and sets of final states. The set of initial states, and the set of final states may be characterized respectively by a precondition Q, and a postcondition R, which are both just predicates on the program variables. Using this model, one systematic way to compose a program in a stepwise manner is by a sequence of refinements, each of which enlarges either the set of initial states or final states (or both) for which the mechanism can establish the postcondition, until finally a mechanism has been composed that will establish the postcondition for the given precondition. This refinement strategy, when combined with rules for manipulating specifications, a means for partitioning the set of initial and final states, and a set of rules for incorporating new refinements, provides a constructive method for program development.

2. AN EXAMPLE

Before describing the method in detail, a simple example will be used to convey the notions, upon which it is based. In the discussion a familiarity with Dijkstra's methodology [3,4] will be assumed. For illustration, the problem chosen is that of finding the maximum m of, and its position p, in a fixed integer array $A[1..N]$. The associated precondition Q and postcondition R with non-fixed free variables p and m are:

$$Q\colon N \geq 0$$
$$R\colon (N=0 \textbf{ cor } 1 \leq p \leq N \wedge m = A_p) \wedge A(j{:}1 \leq j \leq N{:}m \geq A_j)$$

Given a specification (Q,R), before development can proceed, it is necessary for the postcondition R to be in a form where:

(i) it is easy to make an initialization of some non-fixed free variables in R that is sufficient to establish R in some limiting circumstance

(ii) and, having made such an initialization, for some given data configuration, it is easy to check whether or not the initialization has established R.

By "easy" in (i) we mean by a simple assignment or assignments and by "easy" in (ii) we mean by a simple conditional test involving free variables. If the given postcondition is not in this form it should be transformed into an equivalent or related form where (i) and (ii) are satisfied. Rules for doing this are given in the next section. When a postcondition satisfies the requirements (i) and (ii) it is said to be **constructive**. A constructive postcondition is needed to start the development of the program.

In our example, the set of non-fixed free variables is $X = \{p,m\}$. N and A are also free variables but they are **fixed** (i.e. not to be changed by the program). Caps will be used for fixed free variables throughout. For our example, when we attempt to make an initialization of some non-fixed free variables in X, that can establish R we find that R is not in a constructive form. It therefore requires a transformation. A simple state-space extension, introducing another free variable i in place of N can transform R into a form that is constructive.

We get:

$$R_D: i=N \wedge (i=0 \text{ cor } (1 \le p \le i \wedge m=A_p)) \wedge A(j:1 \le j \le i:m \ge A_j)$$

The condition $i=N$ in R_D is referred to as the **equivalence condition**. State-space extensions usually involve an equivalence condition. When all occurrences of i in R_D are replaced by N and the result is simplified, R is retrieved. This amounts to a simple predicate transformation by substitution. The set of non-fixed free variables for R_D has expanded to $X_0 = \{p,m,i\}$. An initialization is now made for the smallest subset x_0 of free variables from X_0 that is sufficient to establish R_D. The assignment $i:=0$ is sufficient to establish R_D, and the equivalence condition $i=N$ is sufficient to determine whether the initialization has established R_D and hence R. A refinement that accommodates the fact that the initialization $i:=0$ may not always be sufficient to establish R for the Q given can be written as follows:

```
$s_0(x_0)$ Initialization $x_0 = \{i\}$
        $i := 0;\ \{P_0\}$
        if $i = N \rightarrow$ skip          $\{R_D$ established$\}$
        [] $i \neq N \rightarrow \{Q_1\}\ s_1(x_1)\ \{R\}$
        fi
```

Using the semantic rule for assignment the weakest precondition for which the initialization $i := 0$ will establish R_D simplifies to:

$$wp(i := 0,\ R_D) = (N = 0)$$

The corresponding set of initial states $\mathcal{A}_0$ for which $s_0(x_0)$ will establish R_D and hence R is:

$$\mathcal{A}_0 = \{A, N \mid N = 0\}$$

The condition, P_0, established by the initialization, $s_0(x_0)$ is:

$$P_0: i = 0 \wedge A(j:1 \le j \le i:m \ge A_j)$$

and the following relations hold:

$$P_0 \wedge i = N \Rightarrow R_D$$
$$P_0 \wedge i \neq N = Q_1$$

The condition corresponding to an initialization of the smallest subset of free variables x_0 in X_0 that is sufficient to establish R_D, at least in some limiting circumstance is referred to as the **base invariant** P_0. The strategy for development, once the base invariant has been determined, is to make subsequent refinements for specifications that correspond either to progressively weakening the base invariant or making a state-space extension by introducing additional free variables. Both these manipulations lead to refinements that can extend the number of initial states or final states for which the mechanism will establish the postcondition. The base invariant P_0, and the **derivative invariants,** P_1, P_2, ... constructed from it, have the same basic form. They characterize the set of **initial states** for which the mechanism constructed from the refinements $s(x_0)$, $s(x_1)$ ... $s(x_i)$ will establish R_D, and the corresponding set of **final states** established by this sequence of refinements. With this form for P_i the goal of the progressive refinement process is to monotonically expand both the set of initial states and/or the set of final states until we have a derivative invariant P_i which, when evaluated on the original state space at least covers:

$$Q \wedge R$$

That is, refinements are made until a mechanism has been constructed which will establish the given postcondition R for a set of initial states that encompasses the given precondition Q.

Returning to our example, since all free variables have yet to be initialized, the next refinement $s_1(x_1)$ may involve an initialization of other free variables that is sufficient to establish R. Such a step, which initializes m and p and extends the range of i (and hence expands the set of initial states for which R can be established) is:

$$p := i+1;\ m := A_p;\ i := i+1$$

Given Q_1, the equivalence condition ($i = N$) can again be used to check whether this latest refinement establishes R_D for a given set of fixed free variables $\{A, N\}$. Including this refinement and accommodating the fact that it may not be sufficient to establish R_D for all initial states defined by Q we get:

$S_1(X_1)$. First Refinement $x_1 = \{p, m, i\}$ †

```
i := 0; {$R_0$}
if i = N → skip
▯ i ≠ N →
        $p := i+1;\ m := A_p;\ i := i+1;\ \{P_1\}$
        if $i = N$ → skip $\{R_D$ established$\}$
        ▯ $i \neq N \rightarrow \{Q_2\}\ s_2(x_2)\ \{R\}$
        fi
fi
```

The derivative invariant P_1 assumes the form:

$$P_1: 0 \le i \le 1 \wedge (i = 0 \textbf{ cor } (1 = p = i \wedge m = A_p)) \wedge \mathbf{A}(j{:}1 \le j \le i{:}m \ge A_j)$$

and the following relations hold:

$$P_1 \wedge i = N \Rightarrow R_D$$
$$P_1 \wedge i \neq N = Q_2$$

Evaluating P_1 on the original state space $\{A, N, m, p\}$ we get the conjunction defining the set of initial and final states for which R is established by the mechanism composed of $s_0(x_0)$ and $s_1(x_1)$ (where $S_1(X_1) = s_0(x_0) \circ s_1(x_1))$).

$$P_1(A,N,m,p): 0 \le N \le 1 \wedge (N = 0 \textbf{ cor}(1 = p = N \wedge m = A_p)) \wedge \mathbf{A}(j{:}1 \le j \le N{:}m \ge A_j)$$

which is not yet equivalent to $Q \wedge R$, and so further refinements are needed. By going one step further and evaluating P_1 on the initial data state space $\{A, N\}$ we can determine the set of initial states for which the mechanism $S_1(X_1)$ will establish R_D i.e.

$$P_1(A,N): 0 \le N \le 1 \wedge \mathbf{A}(j{:}1 \le j \le N{:}A_1 \ge A_j)$$

and therefore the set of initial states for which R_D can be established has been expanded to:

$$\mathcal{A}_1 = \{A,N \mid 0 \le N \le 1 \wedge \mathbf{A}(j{:}1 \le j \le N{:}A_1 \ge A_j)$$

In other words if the data set is empty, or it contains only one element, the mechanism we have will be sufficient to establish the postcondition.

† Note X_1 is the set of free variables defined at this stage in the development and x_1 is the subset of X_1 used to make the refinement $s_1(x_1)$.

To guide the next refinement $s_2(x_2)$, the derivative invariant P_1 needs to be weakened further. One way to do this is to make a change to P_1 that corresponds to allowing some subset of its free variables to be changed. To keep refinements as simple as possible we choose to adopt a form of the **principle of least action** [7]. That is, we will make our refinement by allowing only the smallest subset of free variables to be changed which is sufficient to establish R given Q_2. Increasing just i while holding p and m constant will be sufficient to establish R provided it can be increased to N (refer to equivalence condition). This can be verified by computing wp ("$i := i+1$", R_D). It follows that the precondition P_2 for the next refinement $s_2(x_2)$, obtained by weakening P_1, for the subset $x_2 = \{i\}$ may take the form:

$$P_2\colon 0 \le i \wedge (i = 0 \text{ cor}(1 = p \le i \wedge m = A_p)) \wedge \mathbf{A}(j{:}1 \le j \le i{:}m \ge A_j)$$

This precondition P_2 may serve as an invariant for the corresponding refinement $s_2(x_2)$ which has the responsibility of changing i. Either the loop construction technique described by Gries [4] or Hehner's recursive refinement method [8] may be used to make the refinement. Opting for a loop and employing the technique of forced termination [9] in the implementation we get:

$S_2(X_2)$. Second Refinement $x_2 = \{i\}$

```
i := 0; {$P_0$}
if i = N → skip
[] i ≠ N →
    p := i+1; m := $A_p$; i := i+1; {$P_1$}
    if i = N → skip
    [] i ≠ N →
        n := N;
        repeat {$P_2$}
            if $A_{i+1} \le m$ → i := i+1
            [] $A_{i+1} > m$ → n := i
            fi
        until i = n; {$R_2$}
        if i = N → skip {$R_D$ established}
        [] i ≠ N → {$Q_3$} $s_3(x_3)$ {R}
        fi
    fi
fi
```

The condition established by $s_2(x_2)$ is:

$$R_2\colon i \ge 1 \wedge p = 1 \wedge m = A_p \wedge \mathbf{A}(j{:}1 \le j \le i{:}m \ge A_j) \wedge (i = N \text{ cor } i < N \wedge A_{i+1} > m)$$

Also,

$P_2 \wedge i \ne n \Rightarrow P_2$, $\quad$ $P_2 \wedge i = n \Rightarrow R_2$,
$P_2 \wedge i = N \Rightarrow R_D$ $\quad$ $P_2 \wedge i \ne N \Rightarrow P_2$,

and

$R_2 \wedge i = N \Rightarrow R_D$
$R_2 \wedge i \ne N = Q_3$

Evaluating P_2 on the original state space, as in the previous step we obtain the conjunction $P_2(A,N,m,p)$ defining the expanded set of initial and final states for which R is established by the mechanism composed of $s_0(x_0)$, $s_1(x_1)$, and $s_2(x_2)$ (where $S_2(X_2) = s_0(x_0) \circ s_1(x_1) \circ s_2(x_2)$).

$$P_2(A,N,m,p)\colon 0 \le N \wedge (N = 0 \text{ cor } 1 = p \le N \wedge m = A_p) \wedge \mathbf{A}(j{:}1 \le j \le N{:}m \ge A_j)$$

Again, although P_2 may encompass significantly more states that P_1 it is still not equivalent to $Q \wedge R$, and so further refinements are needed. Evaluating P_2 on the initial data space $\{A,N\}$ gives the set of initial states for which $S_2(X_2)$ will establish R_D i.e.

$$P_2(A,N)\colon 0 \le N \wedge \mathbf{A}(j{:}1 \le j \le N{:}A_1 \ge A_j)$$

and therefore the set of initial states is for which R_D can be established has been expanded to:

$$\mathcal{A}_2 = \{A,N \mid 0 \leq N \wedge \mathbf{A}(j{:}1 \leq j \leq N{:}A_1 \geq A_j)\}$$

In other words, if the data set is empty or the **first** array element is greater than or equal to the rest of the array elements then the composite mechanism $S_2(X_2)$ will be sufficient to establish R_D.

Notice that in the most recent specification the restriction of N to an upper bound of **one** has been removed but that there still remains an upper bound of one on p. The next refinement $s_3(x_3)$ that can establish R needs to accommodate the possibility that the condition $A_1 \geq A[1..N]$ does not always hold. When Q_3 applies, allowing just i, to increase by itself cannot be done without violating P_2, since $s_2(x_2)$ has established the condition $A_{i+1} > m$. To guide the next refinement P_2 needs to be weakened further to admit more initial states for which R can be established. The condition $A_{i+1} > m$ suggests that this can be done by making a change to P_2 that corresponds to allowing the variables p and m to assume values other than 1 and A_1's value respectively. Weakening P_2 accordingly we get:

$$P_3{:}\ 0 \leq i \wedge (i = 0 \ \mathbf{cor}\ (1 \leq p \leq i \wedge m = A_p)) \wedge \mathbf{A}(j{:}1 \leq j \leq i{:}m \geq A_j)$$

The subset of free variables to be changed in making the next refinement $s_3(x_3)$ is $\{p,m,i\}$ which is just the same as the subset that was used in making the refinement $s_1(x_1)$. In fact it is easy to confirm that, given Q_3, the previous refinement $s_1(x_1)$ would be sufficient to make the next refinement $s_3(x_3)$. Our development process has identified what corresponds to a least fixed point. Or, in other words, the development process has uncovered a **cycle** in the structure of the program. We are therefore at liberty to implement the next refinement as either a loop or recursive call embodying the mechanism $s_1(x_1) \circ s_2(x_2)$ with P_3 serving as the loop invariant. Employing a loop and replacing the ***if ... repeat ... until*** construct by the equivalent ***do ... od*** loop we end up with the following mechanism:

$S_3(X_3)$. Third Refinement $x_3 = \{p,i,m\}$

```
{$Q$}
$i := 0;$
do $i \neq N \rightarrow \{P_3\}$
        $p := i+1;\ m := A_p;\ i := i+1;$
        $n := N;$
        do $i \neq n \rightarrow \{P_3\}$
                if $A_{i+1} \leq m \rightarrow i := i+1$
                [] $A_{i+1} > m \rightarrow n := i$
                fi
        od $\{P_3\}$
od
{$R$}
```

where $P_3 \Rightarrow wp(S_3(X_3), R_D)$

and $P_3 \wedge i = N \Rightarrow R_D$
$P_3 \wedge i \neq N \Rightarrow P_3$

Happily this time when we evaluate P_3 on the original state space we get:

$$P_3(A,N,m,p){:}\ 0 \leq N \wedge (N = 0\ \mathbf{cor}\ (1 \leq p \leq N \wedge m = A_p)) \wedge \mathbf{A}(j{:}1 \leq j \leq N{:}m \geq A_j)$$

which has the form $Q \wedge R$ and so no further refinements are required, the development is complete. We now have a mechanism that will establish the postcondition for **all** the initial states.

To complete details for a proof of correctness, P_1, P_2, and R_2 need to be weakened to reflect the possibility that p and m can be changed.

Summarizing, the development of the maximum-finding program has been made by a sequence of refinements, each of which, enlarges the set of initial states for which the mechanism can establish the postcondition until finally a mechanism has been composed which can establish the postcondition for a set of initial states that corresponds to the given precondition for the problem.

3. A MODEL FOR STEPWISE REFINEMENT

The refinement strategy alluded to in the previous section depends on a set of postcondition transformations, a method for partitioning the initial and final states, and another set of rules for incorporating new refinements into an existing program structure. These issues will now be con sidered.

3.1. Postcondition Transformations

A specification given for a problem, although correct, is often not in a form that is directly useful for developing a program which satisfies that specification. For example the postcondition R in the specification:

$$Q: X > 0 \wedge Y > 0$$
$$R: x = gcd(X,Y)$$

which requires that an x be found which is the greatest common divisor of two fixed positive integers X and Y does not give much insight into how the program could be developed. Faced with this sort of situation a good program designer should always be prepared to meet the need to transform a specification into a form that can aid the program's development.

The first problem is recognizing when a postcondition is not in a form that readily aids development of the program. Frequently the criteria suggested in the previous section can signal this situation. That is, if it is not easy initialize a subset of free variables that will establish R, and to check that the initialization has established R then the postcondition needs to be transformed. In making transformations on a postcondition the objective is to introduce more **accessible freedom** into the specification which may subsequently be exploited in the development of the program.

3.1.1. State Space Extensions

A simple way to introduce more accessible freedom into a postcondition is by making a **state-space extension** to give an **equivalent or related form**. This was done with the maximum-finding problem discussed in the previous section. A state-space extension transformation involves replacing one or more constants (or variables) in the original postcondition by new free variables and then making a conjunction of this new form with an equivalence condition. There are several different types of state space extension.

Constant Replacement

In this extension a constant is replaced directly by a variable and the equivalence condition specifies the equality between the constant and the variable.

[1] Given the postcondition

$$R: s = \Sigma(j:1 \leq j \leq N:A_j)$$

it can be transformed to a constructive form by replacing the constant N by the variable i to give

$$R_D: s = \Sigma(j:1 \leq j \leq i:A_j) \wedge i=N$$

where $i = N$ is the equivalence condition.

Variable Replacement

In this extension either a **variable** or an **expression** is replaced by another variable.

[2] The postcondition for finding the integer square root a of a fixed integer N may be given as:

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

Replacing the expression $(a+1)$ by b gives

$$R_D\text{: } a^2 \leq N < b^2 \wedge (b = a+1)$$

where $b = a+1$ is the equivalence condition.

Image Extension

A more interesting form of state space extension involves incorporating a free variable "image" of an existing subexpression within the postcondition. Image extension is a generalization of the idea of replacing a constant by a variable. The following examples illustrate the technique.

[3] Returning to the *gcd* example we had

$$R\text{: } x = gcd(X,Y)$$

This postcondition has the underlying structure

$$x = CONSTANT$$

where the "CONSTANT" is $gcd(X,Y)$ and the free variable is x. To transform this postcondition the constant $gcd(X,Y)$ is replaced by its variable image $gcd(x,y)$, and the equivalence condition needed to establish the equivalence with R is $x = y$, and so we get:

$$R_D \; x = y \wedge gcd(x,y) = gcd(X,Y)$$

We can check that R_D transforms to R as follows. Replacing the new free variable y in R_D by x gives:

$$x = x \wedge gcd(x,x) = gcd(X,Y)$$

which simplifies to

$$x = gcd(X,Y) \text{ **since** } gcd(x,x) = x$$

and so R is retrieved.

[4] There is often another motivation for doing a state space expansion. That is to introduce more freedom into a postcondition in an effort to discover a more efficient algorithm. In the quest for greater efficiency for the exponentiation problem (i.e. $R\text{: } z = X^Y$) we can apply an image transformation by introducing x^y as the image of the constant X^Y. In this case we get

$$z = X^Y \times x^{y'}$$

which can be balanced by rearrangement to give

$$z \times x^y = X^Y, \textbf{ where } y = -y'$$

and after adding the equivalence condition we get

$$R_D\text{: } z \times x^y = X^Y \wedge y = 0 \wedge x = X$$

with $y = 0 \wedge x = X$ acting as the equivalence condition. This new specification admits the possibility of changing X indirectly via x. A doubling strategy can be used for this purpose.

3.2. Decomposition Rules

The partitioning strategy suggested consists of three hierarchical decomposition rules. For a program *S(X)*, described by a free variable set X, and composed by a sequence of refinements $s_0(x_0)$, $s_1(x_1)$, ... $s_N(x_N)$, where $x_i \subseteq X_i$, and X_i is the set of free variables available at the *ith* refinement, the following rules apply:

Primary Decomposition Rule (sufficiency rule)

> *"Each member of the sequence of refinements $s_0(x_0)$, ... $s_N(x_N)$ needed to compose the program S(X) should extend either the set of initial states or the set of final states (or both) for which the postcondition is established beyond that established by all previous refinements."*

Expressed more pragmatically, what the primary decomposition rule suggests is that we should always try to make refinements that will establish the postcondition at least in some limiting circumstance.

The primary decomposition rule provides a basic framework for making refinements. A second, minimum progress rule, based on the **principle of least action** [7], is needed to make the partitioning process more explicit.

Secondary Decomposition Rule (correspondence rule)

> *Given a precondition Q_i, that has previously been established, the next refinement in the sequence $s_i(x_i)$, should be made by assigning values to, or changing, the* ***smallest*** *subset x_i (where $x_i \subseteq X_i$) of free variables which is sufficient to extend the set of initial states or final states (or both) for which the postcondition is established.*

This rule serves two important functions. It provides an explicit rule for refinement in terms of free variables used to define the postcondition, the base invariant, and derivative invariants. It also has the consequence of allowing a match to be made between program control structure and data structure in the sense advocated by Jackson [5,6]. In fact the principle that this rule embodies represents a generalization and abstraction of Jackson's correspondence principle. The rule implies that subsets of free variables and the relations which they describe provide an abstract description of structure. This way of characterizing structure is more powerful than rules based only on semantics as it allows structures for which there is no semantic equivalent to be captured and exploited. What also follows is that if other than a minimum subset of free variables which is sufficient to establish *R* is permitted to change in a given refinement the correspondence between program control structure and data structure is destroyed.

Although explicit, the secondary decomposition rule is not always strong enough to define the minimum subset x_i of free variables which are candidates for change in a particular refinement $s_i(x_i)$ given i. At this point we have the choice of making an arbitrary choice among the candidate minimum subsets of the same cardinality or introducing a third finer decomposition rule. One such rule, which is again an extension of the principle of least action takes the form:

Tertiary Decomposition Rule (simplification rule)

> *For variable subsets of equal size selected by the secondary rule, that subset x_i is selected for the refinement $s_i(x_i)$ which needs to reference but not necessarily change the least number of free variables from X_i*

Of course this rule, like the secondary decomposition rule, cannot always guarantee to yield a unique subset x_i. In such instances an arbitrary choice can be made as to which subset is used for the refinement. More elaborate refinement rules could be included but this would be an unnecessary complication of the method. There are instances where symmetries exist in a problem that can only be dealt with by arbitrary choice. The main function of this tertiary rule is to try to ensure that the simplest refinements are always made first, the tacit assumption being, if more variables are involved, a refinement is more complex. Other criteria may be preferred for simplification.

What has come out of the previous discussion on decomposition is that useful structure in a problem can be characterized by relations defined in terms of variables and conditions applied to particular domains. Furthermore the structure in programs is built from just these materials and so it is reasonable to view decomposition of a problem in terms of changing certain subsets of variables towards their final states under certain conditions. **That is, we need to get used to the idea of viewing the decomposition of a problem in these more concrete terms rather than seeing it as some vague process we are able to perform either by experience, luck, or intuition, by grasping at some vague notion or semantics that seems relevant.** It is true that problems get decomposed and programs get written but it is much less often that problems are, in Plato's words, divided into parts "...at their **joints**, as nature directs, not breaking any limb in half, as a bad carver might". What is important to understand is the virtue of the structure of a solution to a problem that fits or matches the problem. Such programs are clear, concise, consistent, easily changed and hopefully the best solution to a problem.

3.3. Refinement Composition Rules

We can characterize how goal-establishing refinements may be composed by relating each such refinement to the alternative construct which guards the **next** refinement. The rules are very simple; the current refinement $s_i(x_i)$ may be made such that it is either **prefix, infix,** or **postfix** with respect to the alternative construct which guards the next refinement. With R_i defined as the condition that the refinement $s_i(x_i)$ guarantees to establish, and D_i defined such that $R_i \wedge D_i \Rightarrow R$ the composition rules assume the form given below.

(i) PREFIX REFINEMENT

```
*s_i(x_i);*
***if*** *D_i → skip*
*[] ¬ D_i → "Make* ***next*** *refinement capable of establishing R"*
***fi***
```

(ii) INFIX REFINEMENT

```
***if*** D_i → s_i(x_i)
[] ¬ D_i → "Make ***next*** refinement capable of establishing R"
***fi***
```

(iii) POSTFIX REFINEMENT

```
***if*** ¬ D_i → "Make ***next*** refinement capable of establishing R"
[] D_i → skip
***fi***; {D_i}
s_i(x_i)
```

(iv) RECURSIVE OR ITERATIVE REFINEMENT †

$s_j(x_j), \ldots s_{i-1}(x_{i-1})$ where $0 \leq j \leq i\text{-}1$

4. CONCLUSIONS

We claim that inductive stepwise refinement is strongly supportive of goal-oriented programming. The method is a limiting form of top-down stepwise refinement. It uses specifications as a vehicle for guiding problem decomposition and refinement. Refinements are directly linked to the initial and final state spaces. Program development is therefore seen as making a sequence of refinements which enlarge the set of initial states and final states until both the precondition and the postcondition are fully covered. This is in sharp contrast to traditional top-down design where problem decomposition and refinement are seen in much vaguer terms. The principle of least action gives the present refinement strategy specificity and helps

† Implementation of this refinement may be made using either a loop or tail recursion.

it achieve a correspondence between program control structure and data structure. Even more importantly viewing structure in abstract terms rather than via some limited set of semantic tags gives Jackson's correspondence principle much greater generality and wider applicability than the data processing domain. Another lesson to have come out of this study has been the value of transforming specifications to equivalent or related forms. It is hoped that the suggestions that have been made give an added dimension to those tools that are currently employed in this area.

A perceptive criticism of stepwise refinement methods made by Jackson [12] is that "...**we can judge the correctness of a refinement step only by what is yet to come**...". Fortunately this criticism cannot be applied to inductive stepwise refinement because each refinement is made with reference to the postcondition, and hence as far as correctness arguments are concerned, each refinement is independent of what may follow.

Finally what may the program designer expect from this method? Its intended utility will not be gained by blindly applying the method but rather by exploring the idea that a program can be built in a stepwise fashion by creating and combining components each of which is capable of establishing the postcondition. In this context the method can help the program designer to make suitable choices for manageable development steps that will ultimately contribute harmoniously and constructively to a final program that satisfies its specifications. And furthermore, the continuity that the base invariant and its derivatives provide in relating one refinement to the next provides an essential framework for systematic program development.

5. ACKNOWLEDGEMENTS

I would like to thank David Gries, Jifeng He, and Tony Hoare for their helpful and thought-provoking discussions.

6. REFERENCES

1. N. Wirth, "Program Development by Stepwise Refinement", CACM 14, 221-227 (1971)
2. E.W. Dijkstra, "A Short Introduction to the Art of Programming", Report EWD316, Technological University of Eindhoven, August (1971).
3. E.W. Dijkstra, "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, J.N. (1976).
4. D. Gries, "The Science of Programming", Springer-Verlag, N.Y. (1981).
5. M.A. Jackson, "Principles of Program Design", Academic Press, London (1975).
6. M.A. Jackson, "Systems Development", Prentice-Hall, London (1983).
7. A. Mayer, Geschichte des Prinzips der kleinston Action, Leipzig (1877).
8. E. Hehner, "do considered od", Dept. Computer Science Report, University of Toronto (1976).
9. R.G. Dromey, "Forced Termination of Loops", Software Practice and Experience, (in press).
10. R.G. Dromey, "Program Development by Inductive Stepwise Refinement", Software Practice and Experience (in press).
11. G.D. Berglund, "A Guided Tour of Program Design Methodologies", Computer, 14, No. 10, 13 (1981).
12. M.A. Jackson, "Constructive Methods of Program Design", Lecture Notes in Computer Science Vol 44, Springer-Verlag, Berlin (1976).