Derivation of Sorting Algorithms from a Specification

R. G. DROMEY
Computing and Information Studies, Griffith University, Nathan, QLD 4111, Australia

It is important to be able to derive different algorithms that meet a particular specification. Transformations on a program specification provide a systematic means for such an endeavour. Different transformations on a specification can yield new and alternative forms of invariants. These invariants, in turn, can provide the framework for the derivation of a variety of algorithms by the use of weakest precondition techniques. To demonstrate these ideas a number of well-known sorting algorithms are shown to be derivable from a single original program specification.

The intelligent use of equivalent forms is the touchstone of logical insight (S. K. Langer).

Received April 1986, revised September 1986

1. INTRODUCTION

We must learn to use specifications more effectively in program development. The usual scenario for employing them is to define a specification and then manipulate it to aid in the development of a program. Our objective here is to go beyond this scenario and explore the use of a specification as a vehicle for developing a number of alternative solutions to a problem. There are several things that can stand in the way of such an endeavour. Firstly, specifications are often not in a form that will directly permit the development of even a single solution, let alone a number of alternatives. It is therefore necessary to be able to recognise this situation when it occurs and then be able to do something about it.

The problem of recognising that a specification is not in a suitable form for program development has been discussed in detail elsewhere.[1] To summarise, a specification is considered suitable for program derivation if it satisfies the following criteria:

- it is easy to make an initialization of free variables that will establish the postcondition;
- it is easy to check if the initialization has established the postcondition.

Our primary concern here is to develop a number of different solutions to a problem from a single base specification. There are two ways in which manipulations of a specification can influence the development of different algorithmic solutions to a problem. The differentiation is provided by the transformations that are made on a specification, and the order in which those transformations are made in the course of development of a program. We will attempt to show that variability with respect to these two factors can, in some instances at least, lead to a rich variety of strategies for satisfying a given base specification. This variability is possible because a transformation on a specification translates into, in the program domain, a set of constraints under which a particular subset of a program's variables may be changed. And it is the changing of particular subsets of variables, under a particular set of constraints, which ultimately defines the corresponding structure in a program and the strategy or algorithm that will satisfy the specification.

If what we are suggesting is to have any credence, different manipulations of a single sorting specification should lead to the derivation of a number of sorting algorithms. In the sections that follow we shall show how this is indeed the case.

2. TRANSFORMATIONS ON A SORTING SPECIFICATION

A simple formulation of the sorting problem involves the requirement of ordering a fixed set A of N integers using an array $a[1..N]$. The corresponding specification for this problem, phrased in terms of a precondition Q, and a postcondition R, may take the form:

$$Q: N \geqslant 1$$

$$R: (\forall k)(1 \leqslant k < N \Rightarrow a_k \leqslant a_{k+1}) \wedge \text{perm}(a, A)$$

where $\forall$ is the universal quantifier and $\text{perm}(a, A)$ is a predicate indicating that the sorted result must be a permutation of the original fixed set of integers A.

The postcondition R is not in a form where it is possible to make an initialisation of free variables capable of establishing R, that can also be easily checked for success in establishing R. However, making a state-space extension on R, by introducing a new free variable i, we get, by the principle of extensionality:[2]

$$R_D: (\forall k)(1 \leqslant k < i \Rightarrow a_k \leqslant a_{k+1}) \wedge \text{perm}(a, A) \wedge i = N^*$$

where

$$R_D \Rightarrow R$$

Because R_D implies R, it follows that R_D may be used as a basis for program development.

Weakening R_D by dropping the last conjunct we get the invariant

$$P_D: (\forall k)(1 \leqslant k < i \Rightarrow a_k \leqslant a_{k+1})$$

Using Dijkstra's constructive weakest precondition technique[3] and computing

$$wp(\text{`}i := i+1\text{'}, P_D)$$

we are quickly led to the derivation of an insertion-sort algorithm. This derivation is shown in detail elsewhere.[1]

It is difficult to see how we could make any useful transformations on R or R_D that would lead to the derivation of a selectionsort, a bubblesort, a quicksort[4] or any other kind of sort. To make progress we will re-examine the original specification R more carefully. We have:

$$R: (\forall k)(1 \leqslant k < N \Rightarrow a_k \leqslant a_{k+1})$$

The following sequence of manipulations can be made on R and its descendants. Each transformation leads to a

* For proof purposes the range of any free variables (in this case $1 \leqslant i \leqslant N$) introduced may also be added to specifications.

new specification which is at least as strong as its predecessor.

Transformation (I) – introduction of a new free variable q.

$$R^{\mathrm{I}}: (\forall k)(1 \leqslant k < N \Rightarrow (q = k+1 \Rightarrow a_k \leqslant a_q))$$

where

$$R^{\mathrm{I}} \Rightarrow R$$

Transformation (II) – introduction of a quantifier over q.

$$R^{\mathrm{II}}: (\forall k)(1 \leqslant k < N \Rightarrow (\forall q)(k+1 \leqslant q \leqslant N \Rightarrow a_k \leqslant a_q))$$

where $R^{\mathrm{II}} \Rightarrow R^{\mathrm{I}}$.

Transformation (III) – state-space extension, introduction of a new free variable p.

$$R^{\mathrm{III}}: (\forall k)(1 \leqslant k < N \Rightarrow (p = k \Rightarrow (\forall q)\ (k+1 \leqslant q \leqslant N \Rightarrow a_p \leqslant a_q)))$$

where $R^{\mathrm{III}} \Rightarrow R^{\mathrm{II}}$.

Transformation (IV) – introduction of a quantifier over p

$$R^{\mathrm{IV}}: (\forall k)(1 \leqslant k < N \Rightarrow (\forall p)(1 \leqslant p \leqslant k \Rightarrow (\forall q)(k+1 \leqslant q \leqslant N \Rightarrow a_p \leqslant a_q)))$$

where $R^{\mathrm{IV}} \Rightarrow R^{\mathrm{III}}$.

It is also relatively easy to show that

$$R^{\mathrm{IV}} \Rightarrow R$$

and therefore any sorting algorithms that we develop which satisfy R^{IV} will also satisfy R.

There are several ways of expressing the relation R^{IV} with varying degrees of formality, i.e.

$$(\forall k: 1 \leqslant k < N: (\forall p, q: 1 \leqslant p \leqslant k < k+1 \leqslant q \leqslant N: a_p \leqslant a_q))$$

and

$$(\forall k: 1 \leqslant k < N: a[1..k] \leqslant a[k+1..N])$$

These specifications state that for each partition defined by k, the elements in the left partition (i.e. $a[1..k]$) are less than or equal to the elements in the right partition (i.e. $a[k+1..N]$). Quantification over k ensures that the 'sorted condition' expressed in R is still implied by the various versions of R^{IV}.

The postcondition R^{IV} is sufficiently rich in its degrees of freedom to allow manipulations that lead to the development of a variety of different sorting algorithms. We will focus on the last version of R^{IV}, i.e.

$$R^{\mathrm{IV}}: (\forall k: 1 \leqslant k < N: a[1..k] \leqslant a[k+1..N])$$

as the base specification from which to develop sorting algorithms. It should be noted that R^{II} is also a perfectly reasonable specification from which to develop a number of different sorting algorithms.

Before considering the constructive development of any particular sorting algorithm we will provide a brief summary of the main lines of manipulation of R^{IV} that we will use. There are three primary transformations on R^{IV} that we will consider. Each of these primary manipulations leads to a family of sorting algorithms that share an underlying strategy. The three strategies identified are sorting by selection, sorting by insertion, and sorting by partitioning.

2.1 The transformation for selection

An obvious manipulation on R^{IV} we can make is a simple state-space extension in which the leftmost N is replaced by a new free variable i and the accompanying equivalence condition $i = N$ (see Ref. 1) is appended to give:

$$R_D: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..N]) \wedge i = N$$

where $R_D \Rightarrow R^{\mathrm{IV}}$.

The first conjunct in R_D is easily established by the assignments '$i := 1$; $a := A$' while it is much more difficult to establish the second conjunct $i = N$ in concert with the first. The condition established by the initialisation is therefore:

$$P_1: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..N])$$

To make progress towards establishing the remaining conjunct $i = N$ and hence R_D, i will need to be increased. One possibility is to investigate the effect of executing the command '$i := i+1$' under the invariance of P_1. To do this we may compute:

$$wp('i := i+1', P_1)$$

which immediately tells us that the following component is not implied by P_1, i.e.

$$R_1: a[i] \leqslant a[i+1..N] \quad \text{is not implied by } P_1$$

Therefore in order to safely execute the command '$i := i+1$' under the invariance of P_1 it will be necessary to guarantee that the sub-goal R_1 is established first. Pursuit of this goal leads to the derivation of mechanisms that apply a strategy of locating and putting in place, first the smallest element in the array, then the second smallest element in the array, and so on. The particular transformations that are made on R_1 in order to derive a refinement that will establish it determine whether we end up with a direct selectionsort, an indirect selectionsort, a bubblesort, or a heapsort. We will pursue these derivations in the next major section.

2.2 The transformation for insertion

Returning to our original postcondition R^{IV}, a second possible state-space extension is to replace not just one but both occurrences of N by i, and include the equivalence condition $i = N$. This yields

$$R_D: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..i] \wedge i = N$$

where again $R_D \Rightarrow R^{\mathrm{IV}}$.

The first conjunct in R_D is easily established by the assignments '$i := 1$; $a := A$', while it is much more difficult to establish the second conjunct $i = N$ in concert with the first. The condition established by the initialisation is therefore:

$$P_1: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..i]$$

With this specification it is also possible to investigate conditions under which the command '$i := i+1$' can be executed. To do this we may compute

$$wp\ ('i := i+1', P_1)$$

which tells us that

R_1: ordered $(a[1..i]) \leqslant a_{i+1}$ is not implied by P_1*

Therefore in order to safely execute the command '$i := i+1$' under the invariance of P_1 it will be necessary to guarantee that the sub-goal R_1 is established first. In contrast to the previous development, the focus here is not on finding the minimum, and then the second smallest, and so on, but rather on keeping the segment $a[1..i]$ ordered in non-descending order. The particular transformations that are made on R_1 in order to derive a refinement that will establish it determine whether we end up with a direct insertionsort, or an indirect insertionsort. These derivations will also be pursued in the next major section.

2.3 The transformation for partitioning

Both of the previous primary manipulations of the post-condition have focused on building up the sorted or ordered array one element at a time, from the left. Another approach is to be more arbitrary about the order in which the partitioning is done by allowing it to be data-driven. The focus then is upon some arbitrary k value in the range $1 \leqslant k < N$ such that the array is partitioned into two segments $a[1..k]$ and $a[k+1..N]$. In other words the 'for all k' is replaced by 'for one k'. This amounts to weakening R^{IV} by deleting the quantifier to obtain:

$$R_1: 1 \leqslant k < N \wedge a[1..k] \leqslant a[k+1..N]$$

The task is then, in the first instance, to construct a mechanism which achieves the split into two partitions (that is, the array a must be transformed, and a k found, for which relation R_1 is satisfied). This leaves two sub-problems to which the same partitioning strategy can be applied. Depending on how the specifications are manipulated we end up with the derivation of a quicksort algorithm based either on pivot partitioning[5] or interval partitioning.[6] The partitioning algorithm needs to be applied in such a way that it achieves partitioning for all k in the range $1 \leqslant k < N$ in order to guarantee the relation R_D, with its quantification over k.

3. DERIVATION OF SORTING ALGORITHMS

We will now turn to the derivation of some of the sorting algorithms we have alluded to in the previous section. What we will attempt to show is the underlying principles which relate the various algorithms and also the influence of specification manipulations in deriving these algorithms. Space does not permit the complete stepwise development of each one. Examples of detailed stepwise development of algorithms from specifications, in the present style, are presented elsewhere.[1] Fig. 1 shows the basic topology for the various sorting algorithms resulting from resolution of the sub-goals in the way we have outlined.

3.1 The selection algorithms

The first family of sorting algorithms whose derivation we will consider are selectionsort,[4] bubblesort[4] and heapsort.[7] The postcondition that we will use for these derivations is:

$$R_D: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..N] \wedge i = N$$

Earlier we derived the invariant

$$P_1: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..N])$$

and investigated $wp('i := i+1, P_1)$ from which we discovered that the sub-goal R_1, i.e.

$$R_1: a[i] \leqslant a[i+1..N]$$

was not implied by P_1. The point where we left off our discussion was with the requirement that it would be necessary to execute '$i := i+1$' under the invariance of P_1. This corresponds to the following basic sorting strategy:

$i := 1$; $a := A$;
do $i \neq N \rightarrow$
  '*Execute* $i := i+1$ *under the invariance of* P_1'
od

We will now investigate how '$i := i+1$' may be executed under the invariance of P_1.

Direct selectionsort derivation

It is not easy to establish the sub-goal R_1 by an initialisation of free variables. In response to this situation, one way to proceed is to introduce a new free variable j into R_1 along with the equivalence condition $j = N$. This transformation yields:

$$R_1^I: a[i] \leqslant a[i+1..j] \wedge j = N$$

where $R_1^I: \Rightarrow R_1$.

The first conjunct of R_1^I is easily established by the initialisation '$j := i$'. It is however much more difficult to establish this conjunct in concert with the second conjunct '$j = N$'. We will therefore work with the first conjunct of R_1^I as an invariant P_1^I and attempt to establish the second conjunct:

$$P_1^I: a[i] \leqslant a[i+1..j]$$

Our basic sorting loop is therefore decomposed to:

$i := 1$; $a := A$;
do $i \neq N \rightarrow \{P_1\}$
  $j := i$;
  do $j \neq N \rightarrow \{P_1^I\}$

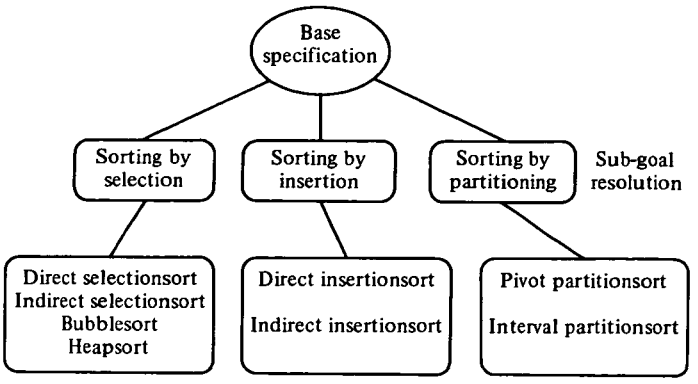

Figure 1. The diagram shows how, at the sub-goal level, specification transformations separate the various sorting algorithms into classes.

* Ordered($a[1..i]$) is shorthand for $(\forall k: 1 \leqslant k < i: a[1..k] r \leqslant a[k+1..i])$. Technically R_1 should have been written as ordered $(a[1..i]) \wedge a[1..i] \leqslant a_{i+1}$, but the intended meaning of the shorthand should be clear.

```
    'Execute j := j+1 under the invariance of P1^I'
  od; {R1^I}
  i := i+1{P1}
od
```

Investigating $wp(\text{'}j := j+1\text{'}, P_1^{\mathrm{I}})$ we find that $a[i] \leqslant a[j+1]$ is not necessarily implied by P_1^{I}. When $a[i] \leqslant a[j+1]$ prevails we can simply execute '$j := j+1$', but when $a[i] > a[j+1]$ occurs we will need to execute swap (a_i, a_{j+1})* together with a j-increment in order to preserve P_1^{I}. Incorporating these refinements we arrive at a description of the main elements of a version of direct selectionsort.

Direct selectionsort

```
i := 1;  a := A;
do i ≠ N → {P1}
  j := i;
  do j ≠ N → {P1^I}
    if a_i ⩽ a_{j+1} → j := j+1
    [] a_i > a_{j+1} → swap (a_i, a_{j+1});  j := j+1
    fi
  od; {R1^I}
  i := i+1
od
```

Indirect selectionsort derivation

To obtain the more often-quoted and more efficient version of selectionsort[4] it is necessary to make two further manipulations of R_1^{I}. First we introduce a new free variable x, in place of a_i, and we record the identity $x = a_i$ as a conjunct to give:

$$R_1^{\mathrm{II}}: x = a_i \wedge x \leqslant a[i+1..j] \wedge j = N$$

We can then go one step further and introduce a new free variable m in place of i in $x = a_i$. In order to do this we need to also add the conjunct $m = i$. This yields

$$R_1^{\mathrm{III}}: x = a_m \wedge x \leqslant a[i+1..j] \wedge m = i \wedge j = N\dagger$$

This relation can be weakened by dropping the conjuncts $m = i \wedge j = N$, which leaves us with the invariant:

$$P_1^{\mathrm{III}}: x = a_m \wedge x \leqslant a[i+1..j]$$

The selectionsort derived using this new loop invariant P_1^{III} is as follows:

Indirect selectionsort

```
i := 1;  a := A;
do i ≠ N → {P1}
  x, m, j := a_i, i, i;
  do j ≠ N → {P1^III}
    if x ⩽ a_{j+1} → j := j+1
    [] x > a_{j+1} → x, m, j := a_{j+1}, j+1, j+1
    fi
  od; {x = a_m ∧ x ⩽ a[i+1..j] ∧ j = N}
  a_m := a_i;
  a_i = x; {R1^II}
  i := i+1
od
```

* Swap is a procedure which swaps its arguments.

† Strictly speaking we should add a conjunct here that indicates that x is a member of the elements $a[i..j]$, i.e. $x \in a[i..j]$.

Bubblesort derivation

For the derivation of a bubblesort we will again work with the same P_1 and R_1 derived in section 2.1, i.e.

$$P_1: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..N])$$
$$R_1: a[i] \leqslant a[i+1..N]$$

Only after first establishing R_1 is it possible to execute '$i := i+1$' under the invariance of P_1. Because we cannot guarantee to establish the sub-goal R_1 directly by an initialisation of free variables, or simply by increasing i, a state-space extension is needed. A suitable extension is to replace both i's by j's and include the equivalence condition $j = i$. This yields

$$R_1^{\mathrm{I}}: a[j] \leqslant a[j+1..N] \wedge j = i$$

where

$$R_1^{\mathrm{I}} \Rightarrow R_1.$$

The first conjunct is easily established by the initialisation '$j := N$'. It may be used as an invariant P_1^{I}, that will allow j to be decreased in order to establish $j = i$. We therefore have:

$$P_1^{\mathrm{I}}: a[j] \leqslant a[j+1..N]$$

The basic structure of the algorithm that follows from this development is:

```
i := 1;  a := A;
do i ≠ N →
  j := N;
  do j ≠ i →
    'Execute j := j−1 under the invariance of P1^I'
  od;
  i := i+1
od
```

Investigating $wp(\text{'}j := j-1\text{'}, P_1^{\mathrm{I}})$ we find that $a_{j-1} \leqslant a_j$ is not necessarily implied by P_1^{I}. When $a_{j-1} \leqslant a_j$ prevails we can safely execute '$j := j-1$', but when $a_{j-1} > a_j$ occurs we will need to execute swap (a_{j-1}, a_j) together with the j-decrement in order to decrease the variant function $j-i$ while preserving P_1^{I}. Incorporating these refinements we arrive at a description of the main elements of a version of bubblesort.

Bubblesort

```
i := 1;  a := A;
do i ≠ N → {P1}
  j := N;
  do j ≠ i → {P1^I}
    if a_{j−1} ⩽ a_j → j := j−1
    [] a_{j−1} > a_j → swap (a_{j−1}, a_j);  j := j−1
    fi
  od; {R1^I}
  i := i+1
od
```

Heapsort

The same underlying selection strategy for sorting is employed in heapsort[7] as in the selectionsorts and bubblesort. All three algorithms repeatedly find the minimum in the unsorted part of the array $a[i..N]$. The difference with heapsort is that it uses a more sophisticated and more efficient method for establishing

$$R_1: a[i] \leqslant a[i+1..N]$$

To carry out the development of this algorithm the insight is needed that a heap provides a very convenient way of finding the minimum in the region $a[i..N]$. The minimum resides at the root of the heap (i.e. at $a[N]$), and only minor adjustments are needed to obtain successive minima from a heap once it has been built. In the first instance the data must be arranged in a heap such that

$$a_{N-j+1} \leqslant a_{N-2*j+1}, a_{N-2*j}$$

holds for all values of j where the indices are within the bounds of the array. This property defines a heap with the minimum at a_N. Using the same P_1 and R_1 as in the last two algorithms we are led to the following implementation of heapsort.

Heapsort

$a := A;$
buildheap $(a, N);$
$i := 1;$
do $i \neq N \rightarrow \{P_1\}$
  swap $(a_i, a_N);$ $i := i+1;$
  siftleft (a, i, N)
od

where *buildheap* is a procedure that builds a heap with its minimum at a_N, and *siftleft* restores the heap property in the array segment $a[i..N]$. The procedure *siftleft* acts as a selection mechanism for successively extracting the smallest element from the contracting region $a[i..N]$ via the root of the heap.

Unfortunately, there is no interesting manipulation of the particular base specification we have used that would lead us directly to the detailed implementation of heapsort. To develop heapsort directly from a specification it is necessary to start with a postcondition that focuses on a tree-like ordering dominance. Such a treatment is provided elsewhere by Dijkstra and van Gasteren.[8]

3.2 The insertion algorithms

The two insertionsort algorithms that we will derive here result from using two different methods for inserting an element at a position in an ordered sequence that preserves the order of the extended sequence. The different methods for insertion arise directly from making different transformations on the sub-goal that must be established in order to extend the ordered sequence by one element.

Direct insertionsort derivation

Earlier in section 2.2 we saw that a second basic form of postcondition to work with was

$$R_D: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..i]) \wedge i = N.$$

From this we were led to the invariant:

$$P_1: (\forall k: 1 \leqslant k < i: a[1..k] \leqslant a[k+1..i])$$

Our investigation of increasing i under the invariance of P_1 led to the discovery that R_1 was not implied by P_1.

$$R_1: \text{ordered}(a[1..i]) \leqslant a_{i+1}$$

In establishing the sub-goal R_1 the obligation is to ensure that $a[1..i]$ remains ordered. Our basic loop structure at this stage is therefore:

$i := 1;$ $a := A;$
do $i \neq N \rightarrow$
  '*Execute* $i := i+1$ *under the invariance of* P_1'
od

As there is no guarantee of being able to establish R_1 directly we may in the first instance seek transformations to another more useful specification. One possible transformation is to introduce a new free variable j to replace the first occurrence of i in R_1 and include the equivalence condition $j = i$ to yield:

$$R_1^{\mathrm{I}}: \text{ordered}(a[1..j]) \leqslant a_{i+1} \wedge j = i$$

where $R_1^{\mathrm{I}} \Rightarrow R_1$.

Dropping the last conjunct from R_1^{I} we obtain the invariant:

$$P_1^{\mathrm{I}}: \text{ordered}(a[1..j]) \leqslant a_{i+1}.$$

Carrying out the necessary loop development using a weakest precondition computation with P_1^{I} we obtain an implementation of direct insertionsort.

Direct insertionsort

$i := 1;$
do $i \neq N \rightarrow \{P_1\}$
  $j := 0;$
  do $j \neq i \rightarrow \{P_1^{\mathrm{I}}\}$
    if $a_{j+1} \leqslant a_{i+1} \rightarrow j := j+1$
    [] $a_{j+1} > a_{i+1} \rightarrow$ *swap* $(a_{j+1}, a_{i+1});$ $j := j+1$
    fi
  od; $\{R_1^{I}\}$
  $i := i+1$
od
$\{R_D\}$

Indirect insertionsort derivation

The more familiar implementation of an insertionsort can be derived by starting out with a different transformation on R_1. The $a[i+1]$ can be replaced by a 'range' over a single element $a[i+1..i+1]$. This yields:

$$R_1^{\mathrm{I}}: \text{ordered}(a[1..i]) \leqslant a[i+1..i+1]$$

where $R_1^{\mathrm{I}} \Rightarrow R_1$.

We can strengthen the right-hand side of the relation further by requiring that it must also be ordered, i.e.

$$R_1^{\mathrm{II}}: \text{ordered}(a[1..i]) \leqslant \text{ordered}(a[i+1..i+1])$$

where $R_1^{\mathrm{II}} \Rightarrow R_1^{\mathrm{I}}$.

Rather than work directly with R_1^{II} we can 'split' the variable i[1] by introducing new free variables l and j to yield:

$$R_1^{\mathrm{III}}: \text{ordered}(a[1..l]) \leqslant \text{ordered}(a[j+1..i+1]) \wedge l = i \wedge j = i$$

where again $R_1^{\mathrm{III}} \Rightarrow R_1^{\mathrm{II}}$.

We can go one step further and obtain

$$R_1^{\mathrm{IV}}: \text{ordered}(a[1..l] \leqslant \text{ordered}(a[j+1..i+1]) \wedge l = j$$

where $R_1^{\mathrm{IV}} \Rightarrow R_1$.

The first conjunct is easily established by the assignments $l := 1;$ $j := i+1$. The second conjunct is much harder to establish in concert with the first

conjunct. Our primary objective now will be to establish R_1^{IV}. For this purpose we will keep the relation

$$P_1^{IV}: \text{ordered}(a[1..l]) \leqslant \text{ordered}(a[j+1..i+1])$$

invariant while attempting to establish $l = j$ primarily by decreasing j and if necessary increasing l using the forced termination technique.[9] Establishing P_1^{IV}, with $l = j$ will be sufficient to imply that R_1 holds. This will allow us to increase i under the invariance of P_1. Carrying out the necessary loop development using weakest precondition calculations with P_1^{IV} we obtain the following implementation of an insertionsort.

Indirect insertionsort.

$i := 1; \quad a := A;$
do $i \neq N \rightarrow \{P_1\}$
  $l := 1; \quad j := i+1;$
  do $l \neq j \rightarrow \{P_1^{IV}\}$
    if $a_{j-1} > a_j \rightarrow swap(a_{j-1}, a_j); j := j-1$
    [] $a_{j-1} \leqslant a_j \rightarrow l := j$
    fi
  od; $\{R_1^{IV}\}$
  $i := i+1$
od

3.3 The partitioning algorithms

The two versions of quicksort that we will derive here result from using two different methods for partitioning a sequence. The different methods for partitioning arise directly from making different transformations on the sub-goal that must be established in order to extend the number of partitions established by one.

Quicksort derivation (using pivot partitioning)

In our earlier discussion of quicksort in section 2.3 we found that the first step was to establish the sub-goal:

$$R_0: 1 \leqslant k < N \wedge a[1..k] \leqslant a[k+1..N]$$

for some arbitrary k value.

As it is difficult to make an initialisation of free variables that will establish R_0 directly we will choose to split the variable k by introducing two new free variables i and j to obtain:

$$R_0^I: 1 \leqslant i < N \wedge a[1..i] \leqslant a[j..N] \wedge i = k \wedge j = k+1$$

where $R_0^I \Rightarrow R_0$.

From R_0^I we can obtain:

$$R_0^{II}: 1 \leqslant i < N \wedge a[1..i] \leqslant a[j..N] \wedge i = j-1$$

The last conjunct of R_0^{II} is difficult to establish in concert with the first two conjuncts by initialisation of free variables. We therefore have the choice of trying to work with this specification or of seeking other transformations. Here we will choose the latter course.

Before proceeding further we will illustrate a new transformation which we refer to as splitting a relation. Suppose, for example, we have the relation:

$$a_i \leqslant a_j.$$

The a_j in this relation can be replaced by x to yield $a_i \leqslant x$ provided $x \leqslant a_j$ also holds. We can then write

$$a_i \leqslant x \wedge x \leqslant a_j \Rightarrow a_i \leqslant a_j$$

Applying a similar transformation to R_0^{II} we obtain

$$R_0^{III}: 1 \leqslant i < N \wedge a[1..i] \leqslant x \wedge x \leqslant a[j..N] \wedge i = j-1$$

where $R_0^{III} \Rightarrow R_0$.

We are free to base our derivation upon R_0^{III}. All but the last conjunct is easily established by the initialisation

$$i := 0; \quad x := a[1]; \quad j := N+1.$$

After adding appropriate limits we will choose as our invariant

$$P: a[l..i] \leqslant x \wedge x \leqslant a[j..u].$$

This is very close to the familiar invariant that is usually employed to develop a partitioning algorithm which serves as the central computational element of quicksort.[5] Below we provide an implementation of quicksort that employs both binary recursion and tail recursion,[10] that is based on the specifications given and the associated weakest precondition calculations for '$i := i+1$' and '$j := j-1$'.

Quicksort

quicksort (**var** *a*:*n elements*; *l*, *u*: *integer*);*
var *i*, *j*, *x*: *integer*;
  $i := l-1; \quad j := u+1; \quad x := a[l];$
  if $i < j-1$ **then** $\{P\}$
    partition:
      do $a_{i+1} < x \rightarrow i := i+1$ **od**;
      do $a_{j-1} > x \rightarrow j := j-1$ **od**;
      if $i < j-1$ **then** $swap(a_{i+1}, a_{j-1}); \quad i := i+1;$
        $j := j-1;$ *partition* **fi**;
    quicksort (*a*, *l*, *i*);
    quicksort (*a*, *j*, *u*)
  fi
end

In carrying out this derivation we chose to introduce two new free variables i and j. We could have arrived at essentially the same implementation by the introduction of just one variable j to replace $k+1$ in R_0. Although the invariant states '$a[1..i] \leqslant x$' should be maintained, the stronger relation $a_{i+1} < x$ is used in the loop implementation. The reason for employing the stronger loop guard $a_{i+1} < x$ is to avoid the termination problems that the weaker guard $a_{i+1} \leqslant x$ would introduce. The stronger guard is also commonly used in most published implementations of quicksort. The same reasoning applies to the guard involving j.

Quicksort derivation (using interval partitioning)

There is an interesting alternative to the pivot partitioning method that we have just derived. To see how it may be obtained we will also start out with:

$$R_0^{II}: 1 \leqslant i < N \wedge a[1..i] \leqslant a[j..N] \wedge i = j-1.$$

Again it is necessary to use a relation-splitting technique. To understand how the technique is applied consider again the relation

$$a_i \leqslant a_j.$$

* This implementation establishes $i \geqslant j-1$ rather than strictly $i = j-1$. Also other versions take more elaborate steps to choose x. Such issues have not been of concern here. Our relations should also indicate that x is a member of the array.

From this relation we can move to

$$a_i \leqslant x \wedge x \leqslant a_j.$$

We can then make a further state-space extension by replacing the x in $x \leqslant a_j$ by a new free variable y which must satisfy $x \leqslant y$. We can then write:

$$a_i \leqslant x \wedge y \leqslant a_j \wedge x \leqslant y \Rightarrow a_i \leqslant a_j.$$

Applying essentially the same manipulations to R_0^{II} yields

$$R_0^{III}: 1 \leqslant i < N \wedge a[1..i] \leqslant x \wedge y \leqslant a[j..N] \wedge x \leqslant y \wedge i = j-1.$$

Dropping the 'hard-to-establish' conjunct and adding limits we arrive at the invariant

$$P_0^{III}: 0 \leqslant i < N \wedge a[l..i] \leqslant x \wedge y \leqslant a[j..u] \wedge x \leqslant y.$$

This invariant when used to make weakest precondition computations for increasing i and decreasing j leads to what is called an interval partitioning algorithm.[6] An implementation of quicksort that employs interval partitioning is:

```
quicksort2 (var a:n elements; l,u: integer);
var i,j,x,y: integer;
  i := l;  j := u+1;
  if i < j-1 then
    if a_i ≥ a_{j-1} then swap(a_i, a_{j-1}) fi;
    x := a_i;  y := a_{j-1};  j := j-1;
    intpartition: {P_0^III}
      do a_{i+1} < x → i := i+1 od;
      do a_{j-1} > y → j := j-1 od;
      if i < j-1 then
        if a_{i+1} ≥ a_{j-1} then swap(a_{i+1}, a_{j-1}) fi;
        i,j,x,y := i+1, j-1, max(x, a_{i+1}), min(y, a_{j-1});
        intpartition
      fi;
      quicksort2(a, l, i);
      quicksort2(a, j, u)
  fi
end
```

4. CONCLUSIONS

We have used manipulations of a specification to derive implementations for a variety of well-known sorting algorithms. The investigation has identified specific transformations that separate the well-known sorting algorithms. A number of these transformations should be applicable to other problems. The techniques that have been used here should also complement the more formal techniques employed elsewhere by Darlington.[11]

The primary objective of this exercise has been to show the potential that specification manipulations have for guiding and exploring different possible solutions to problems. If the family of techniques that we have used here are more widely applicable they raise the prospects of a reasonably systematic strategy for exploring different solutions to well-specified problems. It is hoped that this demonstration may be sufficient to convince others to explore approaches along these lines.

There are several broad observations to come out of this investigation. The first is that we need to go beyond the strategy of simply taking a postcondition and weakening it directly in some simple way.[3] What the present discussion shows is that it is sometimes necessary to go through a considerable strengthening of a specification by introducing new free variables and accompanying conditions before arriving at a form suitable for weakening to obtain an invariant. The process of strengthening a specification is really one of enriching it so that subsequently there are more alternatives to explore when applying weakest precondition techniques.

The second important thing to observe is that weakest precondition calculations may serve as a tool for guiding and ordering the decomposition of a problem. They can do this by identifying parts of a specification that may not be satisfied under the influence of a change in certain variables. What the present examples show is that weakest precondition investigations can identify larger and more complex components of a program as well as the simple components of a loop body that they are usually used to identify. The refinement process that the weakest precondition investigations propagate is accompanied by the stepwise introduction of new free variables which in turn need to be changed under some invariant condition. Different choices made in taking these steps lead naturally to the derivation of different algorithms. There is some pedagogical merit in explaining the differences among sorting algorithms in these terms even though the algorithms were obviously not originally derived in this way. Discovering new and different solutions to problems remains a central and difficult concern for computing science. Techniques that have any potential for showing some 'systematic' light in the right direction therefore need to be investigated or at least put to the test.

Acknowledgements

I would like to thank Martin Bunder, Barbara Davidson and Michael Shepanski for helpful discussions, and David Gries for pointing out reference 11. I would also like to thank the anonymous referee for the constructive comments given.

REFERENCES

1. R. G. Dromey, 'Systematic program development'. *IEEE Trans. on Software Eng.* (in press).
2. P. Suppes, *Introduction to Logic*. Van Nostrand, Princeton, N.J. (1957).
3. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J. (1976).
4. N. Wirth, *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, N.J. (1976).
5. C. A. R. Hoare, Quicksort. *The Computer Journal* **5**(1), 10–15 (1962).
6. M. van Emden, Increasing the efficiency of Quicksort. *Comm. ACM* **13**, 563–567 (1970).
7. J. Williams, Heapsort (Algorithm 232). *Comm. ACM* **7**(6), 347–348 (1964).
8. E. W. Dijkstra and A. J. van Gasteren, An introduction to three algorithms for sorting in situ. *Inf. Proc. Letts.* **15**(3), 129–134 (1982).
9. R. G. Dromey, Forced termination of loops. *Software – Practice and Experience* **15**, 29–40 (1985).
10. E. C. Hehner, do considered od. *Acta Informatica* **11**, 287–304 (1979).
11. J. Darlington, A synthesis of several sorting algorithms. *Acta Informatica* **11**, 1–30 (1978).