6 MWDS and the theory of computation

As described in the introduction to the thesis, this chapter sets out to illustrate how MWDS can potentially be viewed as providing empirical roots for logical constructions. This agenda is only appropriate if the distinction between MWDS and classical programming is presumed to be fundamental in character. For convenience, this chapter is written from this perspective and is acknowledged to make controversial claims.

There are two quite different ways to interpret computer use. The classical theory of computation accounts for interaction with computers in terms of algorithms and formal models of state and automata [HU79]. The semantics of MWDS has an entirely different basis in which the key emphasis is upon the computer as a physical artefact and on experientially mediated interpretation of human-computer interaction (cf. Section 2.2). Beynon [Bey99] argues the case for a non-logicist foundation for AI that stems from such experientially mediated interpretations, and discusses the significance of languages and logics within this framework. This chapter develops a practical case study to demonstrate the relationship between MWDS and the classical theory of computation explicitly.

The key concept in our case study is the ‘data structure’. A data structure serves two purposes: it can be viewed as a physical artefact that metaphorically represents computer manipulation of data; it is also used to frame state transitions within a computer program (cf. Section 5.3.1). Our specific focus in this chapter is upon the *heap* data structure and its application in the heapsort algorithm. The two main sections of the chapter address:

- the use of MWDS to construct a model of the heap data structure as a physical artefact (such as a lecturer might draw on a blackboard when introducing heapsort);

- how a formal account of heapsort, based on Dijkstra's Weakest Precondition (WP) formalism can be interpreted in terms of the definitive Heap model[1].

6.0 Overview

Section 6.1 introduces the heap structure and the conventional heapsort pseudo code. This is followed by a discussion of the development of the Heap model based on the observation-oriented and agent-oriented paradigm provided by MWDS. The model building is closely linked to the kind of experimental analysis that might have led to the discovery of heapsort, and the model is gradually developed in a manner similar to that discussed in [SRCB99]. The definitive Heap model supplies the basis for a non-standard implementation of heapsort, centred on the representation of state as observed and experienced by the modeller. This can be viewed as an ADM Heapsort device that has different characteristics from a conventional heapsort program.

Section 6.2 relates the experiential view of heapsort associated with the ADM Heapsort device to a formal specification of the heapsort algorithm using Dijkstra's WP formalism [Dij76]. The integration of informal and formal views of heapsort within the Heap model is addressed in subsection 6.2.1, where we show how the states in the heapsort process that are characterised by pre- and post-conditions in WP can be interpreted and visualised in the Heapsort device. The characteristics of the Heapsort device as a non-standard model of heapsort are discussed in subsection 6.2.2. In the final subsection (6.2.3), we consider the relationship between MWDS and the classical theory of computation, using heapsort as a case study. This involves construing the different ways in which heapsort is performed by the ADM Heapsort device and by a conventional heapsort program.

6.1 Introduction to the heapsort algorithm

Heapsort is one of the classic sorting methods, which was proposed by Williams [Will64] in 1964. The heapsort process is closely related to TREESORT introduced by Floyd [Floyd64] because the process uses the concept of a tree structure to perform sorting. It can sort an array of N elements in place in $O(N \log N)$ steps [SS93]. Its run-time is competitively fast compared to other sorting algorithms such as Bubblesort, which requires $O(N^2)$ steps.

[1] Some material in this section is drawn from [BRS00]

In the heapsort process, all procedures operate on single word items, stored as elements indexed from *1* to *N* in an array *a*. The elements of the array can be represented by a binary tree (cf. Figure 6-1). In this representation, the element *a[1]* is at the root of the tree and the left and right children of the tree node associated with the element *a[i]* are associated with the elements *a[2i]* and *a[2i+1]* respectively. A binary tree with values attached to its nodes is a *heap* [SS93] if the value attached to each node is greater than or equal to the values attached to its children. Within such a tree, a node is said to be *heap satisfied* (or equivalently to satisfy the *heap condition*) if the value attached to it is greater than or equal to the values attached to its children. If we identify the array *a* with its representation as a binary tree, we can regard *a* as a heap if the elements in the array *a* are such that $a[i] \geq a[j]$ for $2 \leq j \leq N$, $i = j/2$. If *a* is a heap, then *a[1]* is the greatest element of the array. In studying heapsort, the dual order relation on array values defined by $x \leq' y$ if and only if $x \geq y$ is of interest.

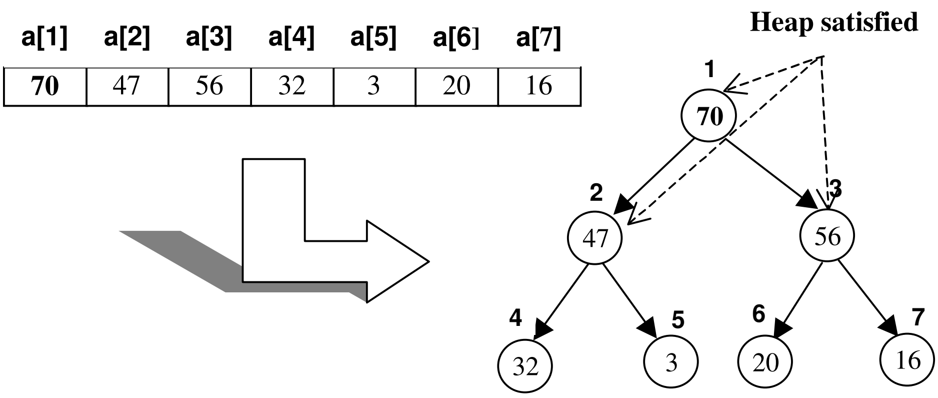


Figure 6-1: An array as a heap

Figure 6-1 illustrates how the heap data structure serves as a physical artefact. The characteristics of the heap structure, as depicted in Figure 6-1, and as interpreted by an observer, are not accurately reflected in the standard computer implementation of the heap data structure. For instance, a student following an exposition of the heap concept based on Figure 6-1 as a blackboard diagram will interpret 'exchanging the elements *a[3]* and *a[6]*' as 'violating the heap condition at node 3' and 'rendering the array *a* no longer a heap' at one and the same time. That is to say, the properties of the heap as a physical artefact enable the observer to experience these three state changes as one (cf. Section 6.1.2).

6.1.1 Conventional implementation of heapsort

The heapsort algorithm is conventionally implemented as a procedural program. Heapsort works in two phases. First, it arranges the elements being sorted into a *heap*, a complete binary tree in which the value of each node is larger than the values of each of its children. Second, it repeatedly removes the root (i.e. the largest value among the elements) from the heap, sets it aside, and reestablishes the heap property, repeating this until the heap is empty.

Box 6-1 displays the pseudo code for the heapsort process as it applies to an array a with N elements. As explained above, the array a is interpreted as a tree by placing the root node at position 1, and for each node at position i, placing its left child at position $2i$, and its right child at position $2i+1$. The heapsort method is based on maintaining a heap on the array values.

```
proc Heapsort {
        n:=N;
        k:=(N+1)/2;                            # Heap establishment
                while (k≠1) {
                        k:=k-1;
                        Shuffle(k,N);
                }                              # Array a is heap (at this point)
                while (n≠0) {                  # Sort extraction
                        a:=swap(1,n);          # swap the element 1 and n
                        n:=n-1;                # reduce the size of heap
                        Shuffle(1,n);          # repair the heap
                }
}

proc Shuffle(lo,hi) {
   k:=lo;                  # CAND is lazy AND evaluation and COR is lazy OR evaluation
   while ((left(k)≤hi) CAND (a[k]<a[left(k)])) COR ((right(k)≤hi) CAND (a[k] < a[right(k)]))
   {      if (right(k)>hi) COR (a[left(k)]≥a[right(k)])      # where left(k) is the left child of k
                then m:=left(k);                              # and right(k) is the right child of k
        else if (right(k)≤hi) CAND (a[left(k)]≤a[right(k)])
                then m:=right(k);
        end if
        a:=swap(k,m);
        k:=m;
   }
}
```

Box 6-1: The pseudo code for the heapsort process

The pseudo code shown in Box 6-1 illustrates that the heap is built from the bottom-up by calling the subroutine **Shuffle**(*lo*, *hi*). The **Shuffle** procedure goes through every parent (i.e. $a[k]$) node and examines whether such a node is heap satisfied; if not, then it can be made heap satisfied by swapping the element of the parent node with the element of one or other of its children, whichever has the greater value.

Once the entire tree is heap satisfied, the greatest element is at the root. The heap extraction process is progressed by swapping the element at the root with the last element of the tree and then reducing the index n (initially N) by 1. This reduces the size of the tree by 1 so that it now becomes $n-1$. At this stage, the entire tree may not be heap satisfied. The **Shuffle** program is called to make the heap satisfied. The process is repeated until n equals 1 and the elements in the array a are then sorted.

The heapsort program specified in Box 6-1 can be regarded as a program device (in the sense introduced in Chapter 5) that outputs a set of elements (input by the user) in sorted order. In particular, it has those characteristics of a device identified in Section 5.3.1 that are associated with its status as an efficient implementation of sorting which (e.g.) minimises the number of observations and comparisons of the elements to be sorted. MWDS supplies an alternative approach to describing the heapsort process, which focuses on representing the observables that capture the state of the heap with reference to dependency and agency. This will be discussed in the next section.

6.1.2 The structure of the definitive Heap model

MWDS approaches the representation of state and algorithms in an unusual way. An algorithm is interpreted as a generic pattern of reliable interaction amongst agents operating within a space for agent action framed by a definitive representation of state (cf. Figure 2-17). When we interpret heapsort in this way, the heap data structure forms an integral part of the representation of the state that is the focus for state changing and observation by all the relevant agents. This is the motivation behind the development of the definitive Heap model. It is helpful to conceive the entire development of the Heap model and its application to heapsort as the elaboration of a construal of agent interaction following an appropriate pattern within an appropriate action space such as depicted in Figure 2-17.

Figure 6-2 depicts the definitive Heap model that is derived by observing the way in which the array and heap data structures in Figure 6-1 can be used in combination as a physical artefact in teaching students the concept of a heap. In this context, the heap viewed as a physical artefact supplies the referent for the MWDS activity. The model is developed by identifying the observables and dependencies to which the students' attention needs to be drawn. These include the way in which array elements correspond to tree nodes, the significant order relations between elements at parent-child (but not child-child) nodes of the tree, and the status of the heap condition at each node. In this context, Figure 6-1 is not to be interpreted as a static diagram such as might appear in a textbook, but as a blackboard sketch to be interactively revised and annotated by the lecturer so as to draw out all the key observables and interactions that are involved in understanding the concept of a heap. As discussed in Section 2.2.1 in connection with

Figure 2-10, the difference between the student and the lecturer in their relation to Figure 6-1 is in their implicit knowledge of the relevant interactions and observables.

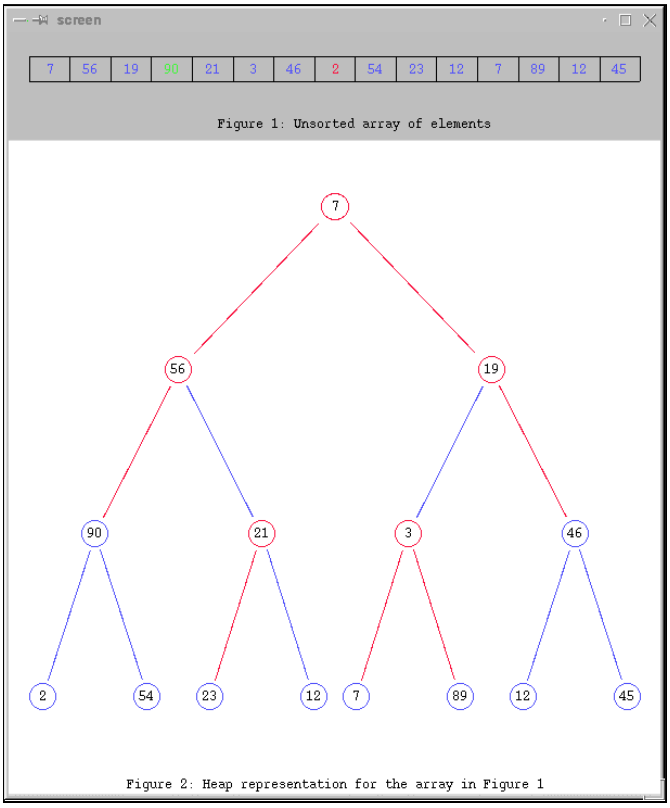


Figure 6-2: A screenshot of the definitive Heap model

Describing heapsort in terms of interactions with the definitive Heap model leads to an abstract state-based model of the algorithm that is more easily apprehended and interpreted than a traditional program. In a conventional implementation of heapsort, the actual state changing activity exhibited by the program is more complex, and involves intermediate states that have no interpretation in the abstract view. For instance, when two values have been exchanged, a procedure will typically have to be invoked in order to update the status of the heap structure. The intermediate state that exists prior to this update is an artefact of the implementation that has no counterpart in the abstract description of heapsort [Bey98a, BRSW98].

The definitive Heap model as depicted in Figure 6-2 is open to interaction outside the pattern of state changes associated with the heapsort algorithm. For instance, the modeller can exchange any pair of elements, can substitute any value at any node and relocate the position of nodes in the tree arbitrarily. This reflects the character of the space for agent action depicted in Figure 2-17, which remains open even though the definitive Heap model is embedded in its state. Further elaboration and extension of the Heap model is required to frame the patterns of agency and state change that are followed in heapsort. Three stages in developing the heap model will be investigated in the context of this extension:

1. Constructing a visual model of a heap with which the user can experiment in order to understand the heap concept.
2. Constructing state-based models to represent the stages in the heapsort process, allowing the user to trace the steps involved in heap-building and sort extraction through a sequence of manual operations.
3. Introducing automatic mechanisms to carry out the appropriate sequence of steps.

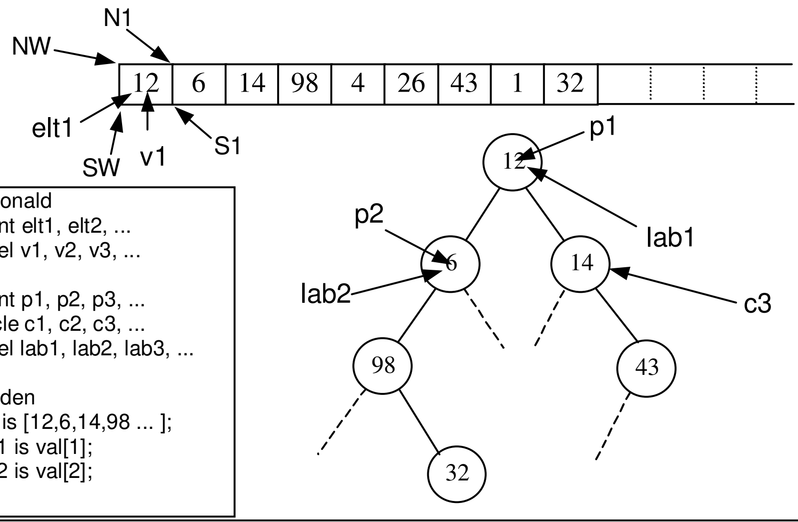
It would be possible to give an account of the model construction that deals with issues 1, 2, and 3 in sequence, but this is not the most appropriate method of organisation to adopt. From the MWDS perspective, the most effective way to present the model construction is to introduce the underlying concepts systematically as they might have been encountered in the discovery of the heapsort algorithm. For instance, a suitable account of heapsort addresses the following issues in turn:

- Building the basic definitive Heap model (cf. Figure 6-2).
- Identifying techniques for establishing the heap condition at every node of the tree.
- Generalising the heap representation to associate array intervals with tree segments.
- Identifying the pattern of heap transformation that is characteristic of heapsort.

The development of the model can accordingly be divided into four distinct parts. These are first: the construction of a visual model of a heap, second: the establishment of the heap condition (control and agency), third: heap generalisation (associating tree segments with array intervals) and fourth: the construction of state-based models of the heapsorting process.

Constructing a visual model of a heap

It is necessary for the user[2] who inspects the model to first understand the relationship between the disposition of elements in the array and the geometry of the associated tree. In Figure 6-2, the first element in the array is represented by the root of the tree, and the elements indexed by $2i$ and $2i+1$ in the array are represented as the left and right children of the node, representing the array element indexed by i. The user can interact with the model manually. To make this correspondence accessible to the user, a dependency is established between the values of the array and the elements of the tree, so that changing the value of an array element simultaneously changes the value of the corresponding node.




```
%donald
point elt1, elt2, ...
label v1, v2, v3, ...
...
point p1, p2, p3, ...
circle c1, c2, c3, ...
label lab1, lab2, lab3, ...
...
%eden
val is [12,6,14,98 ... ];
val1 is val[1];
val2 is val[2];
...
```

```
%donald
elt1 = (SW+N1) div 2                # (NW, SW), (N1, S1), (N2, S2) are pairs of
elt2 = (S1+N2) div 2                # endpoints for vertical lines in the array
...
p1 = {500, 900}
p2 = {p1.1-3*wid, p1.2-len}         # wid and len are the horizontal and vertical
p3 = {p1.1+3*wid, p1.2-len}         # separation distances between tree nodes
...
c1 = circle(p1, radius)
c2 = circle(p2, radius)             # radius is the radius of the circle at each node
...
%eden
_v1 is label(str(var1), _elt1);   /* the definition for a DoNaLD label variable v1 */
_lab1 is label(str(val1), _p1);   /* the definition for a DoNaLD label variable lab1 */
…
```

Figure 6-3: The geometrical representations of the array and the associated tree

Figure 6-3 illustrates how the geometric components of the visual construction are defined. The DoNaLD variables such as elt1, elt2 and v1, v2 define the visual representation of the geometrical elements in the array and the variables such as p1, p2, c1, c2 and lab1, lab2 define the visual representation of the geometrical elements in the tree.

The visualisation of the value of elements both in the array and the tree is linked through variables such as val1, val2 and val3 as shown in Figure 6-3. The values of the label variables v1, v2 and v3 (respectively, lab1, lab2 and lab3), which are used to visualise the value of elements in the array (respectively, in the tree) depend upon the values of the variables val1, val2 and val3.

---

[2] In this context, the neutral term 'user' is adopted to refer to human interpreters playing a number of

To model observations of this nature in a definitive script requires variables to represent the index and value of each node, to record the order relationship (<, =, >) on each edge of the tree, and to register whether the heap condition holds at each node. There is a set of variables to maintain these relationships. For instance, the following are examples of definitions to record the state of the heap condition at each node.

```
hc1 is (val1 >= val2) && (val1 >= val3);
hc2 is (val2 >= val4) && (val2 >= val5);
hc3 is (val3 >= val6) && (val3 >= val7);
```

In MWDS, additional dependencies can readily be introduced to establish suitable visual conventions for representing the abstract conditions. For instance, the circle representing each node is coloured to reflect whether or not the heap condition is satisfied, and the lines connecting nodes are coloured to reflect the order relationship between them (<, =, >). The following definitions are used to define the colour of circles so that they indicate the status of the heap condition at each node.

```
A_c1 is “color=” // hc1? BLUE: RED;
A_c2 is “color=” // hc2? BLUE: RED;
A_c3 is “color=” // hc3? BLUE: RED;
```

Blue circles indicate heap satisfied nodes and circles are otherwise red.

Establishing the heap condition: Control and Agency

At this stage of development of the Heap model, the user can manually exchange the *i*-th and *j*-th elements in the array via a simple procedure exc(*i*, *j*) in order to explore heap building strategies. This manual interaction can reflect different degrees of understanding of the heap concept. For instance, with the visualisation convention for the heap condition in place, the user can learn how to establish a heap by following a set of abstract rules based solely on the perceived colours of nodes and edges in the diagram.

Because user interaction with the model is unrestricted, it addresses issues broader than those encountered in a narrow study of the heapsort procedure as it appears in a conventional program. A user, who wants to understand heapsort can explore what it means for a tree to be a heap by assigning arbitrary values to some variables. A simple technique used to establish the heap condition at every node is to repeatedly exchange the value at a node with that of a child that has a greater value, until no such exchange can be made. A set of definitions to record the

---

different roles such as understanding the definitive Heap model or learning about heapsort index of the child with a greater value at each node *i* can also be introduced. Examples of appropriate definitions are as follows:

```
ixgtch1 is (val2>val3)? 2: 3;       /* the index of the child with the greater value */
ixgtch2 is (val4>val5)? 4: 5;
ixgtch3 is (val6>val7)? 6: 7;
```

The introduction of such observables is associated with issues of control and agency that are complementary to the simple state of model. Such observables are concerned with a particular goal for manipulation of the tree model, for instance establishing the heap condition at node *i*. They can be used to frame the actions of automatic agents at a later stage in the development of the model.

The above discussion is primarily concerned with the construction of the definitive Heap model. The next section will consider the further refinement and extension of the model for application in the heapsort process.

6.1.3 Introducing agency to the model

In order to refine the model for application in the heapsort process, it is necessary to deal with the intermediate structures that arise in building and extracting elements from heaps. This involves introducing bounds on the segment of the array within which the heap condition is assessed.

Heap generalisation: Associating tree segments with array intervals

Two observables are defined as bounds (first and last). Definitions in the model are modified to take account of whether the indices of nodes referenced lie between these two bounds. A new set of definitions for determining whether the indices lie in the predefined range is added as follows:

```
inhp1 is (first<=1)&&(1<=last)? 1: 0;    /* does index 1 lies between first and last? */
inhp2 is (first<=2)&&(2<=last)? 1: 0;
inhp3 is (first<=3)&&(3<=last)? 1: 0;
```

The definition of the heap condition, and of the child with greater value at node *i* then have to be revised and modified as follows:

```
hc1 is (!inhp1)|| (inhp1 && (!inhp2 || (inhp2&&(ord12>=0)) && (!inhp3|| (ord13>=0))));
hc2 is (!inhp2)|| (inhp2 && (!inhp4 || (inhp4&&(ord24>=0)) && (!inhp5|| (ord25>=0))));
ord12 is (val1>val2)?1 : (val2<val1)?-1: 0; /* compare the value between node 1 and2 */
…
ixgtch1 is (!inhp3)? 2: (val2 > val3)? 2: 3;
ixgtch2 is (!inhp5)? 4: (val2 > val3)? 4: 5;
```

The attributes of edges and nodes are likewise modified to reflect membership of the heap:

```
A_c1 is "color=" ((inhp1)? ((hp1)? BLUE: RED): WHITE);
```

All these modifications can be stored in a separate file and included at any point to transform a simple Heap model to a more complex variant. (This may be especially useful when trying to trace errors in scripts, making it possible to carry out experiments in two different contexts.)

One of the advantages of MWDS is that it allows experimentation with a wide variety of objectives. It is possible to explore the effect of changing the range of the heap and the values at nodes both in order to test that the correct definition of the heap condition has been developed, and in order to confirm that the concept of restricting the heap to an array interval is correctly understood.

Constructing state-based models of heapsort process

Once the model of the heap described above has been constructed, it becomes possible to manually trace the steps in the heapsort algorithm. The algorithm consists of a two-stage process as depicted in Box 6-1. The first stage is the establishment of the heap. In this context, the observable first, which ranges from 8 down to 1, is the counterpart of the variable *k* in Box 6-1. The second stage is the sort extraction. In this context, the observable last, which ranges from 15 down to 1, is the counterpart of the variable *n* in Box 6-1. The extraction of an element from the heap is performed by decrementing the observable last and exchanging the value indexed by the former value of last with the value at the root (see the action outsort in Box 6-2).

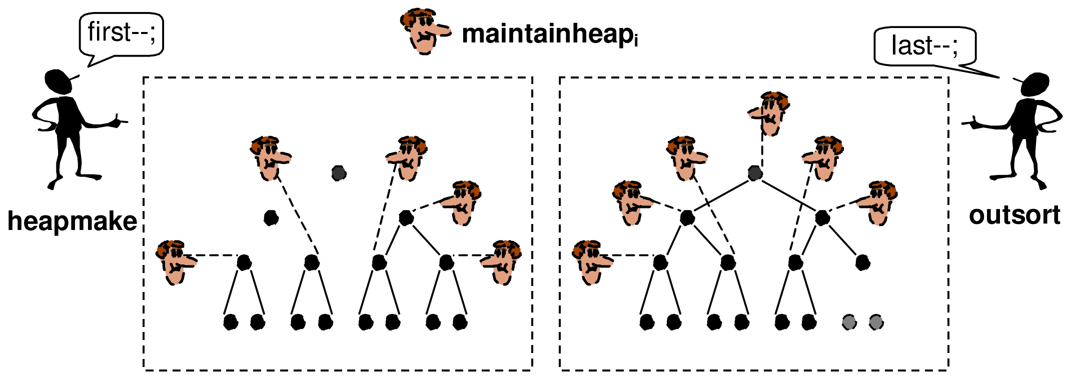


Figure 6-4: Agents to automate heapsort

The model of the heap at this stage admits all the transformations that the user needs in order to simulate the heapsort algorithm by a systematic process of exchanging values at nodes following a predetermined protocol, and changing the range of indices in the heap under consideration. The introduction of agents to perform the heapsort process automatically is of interest. In introducing this automation, the idea is to define an agent to maintain the heap condition at each node and two extra agents to decrease the values of the first and last variables (cf. Figure 6-4).

Figure 6-4 can be viewed as a multi-agent construal of heapsort as a pattern of interaction within an action space of the kind depicted in Figure 2-17. The agents (symbolised by human faces and human figures as in Figure 6-4 and corresponding to agents such as **A** and **B** in Figure 2-17) are construed as monitoring the observables assigned to them. If the values of these variables are not appropriate, the agent takes action to adjust them. This can lead to the propagation of changes. When one agent performs an action, this may affect another agent's observation so that this agent has to adjust its observables which may in turn affect another's state. The scripts for these agents depicted in Figure 6-4 can be seen in Box 6-2.

```
proc maintainheap1: hc1, next {
        if (!hc1 && next==1) {              /* exc will swap the */
                exc(1, ixgtch1);            /* element at position (e.g. 1)*/
                next=0;                     /* and another position (e.g. ixgtch1) */
        }
}
proc maintainheap2: hc2, next {
        if (!hc2 && next==1) {
                exc(2, ixtch1);
                next=0;
        }
}
...
proc nextstep: heapwin_mouse{
        if (heapwin_mouse[2]==4 && next==0) next++;
}
proc heapmake: next {
        if (next && first>1 && is_heap) {   /* initially first=8 */
                first--; next=0;            /* and last=15 */
        }                                   /* is_heap checks the heap condition for an entire */
}                                           /* tree corresponding to the value of first and last */
proc outsort: next {
        if (first==1 && next && is_heap) {
                last--; exc(1, last+1);
                next=0;
        }
}
```

Box 6-2: Eden actions to represent the automatic agents in heapsort

In Box 6-2, the agent maintainheap_i has the task of making the i-th node heap satisfied by exchanging the value at node i with the value at the node ixgtch_i. The triggering variables, such as hc1, hc2 and next, are signals for each agent. If the heap condition at a particular node is not satisfied, the agent which is responsible for that node has to act to make the node heap satisfied again.

The agent heapmake takes care of the establishment of the heap and outsort does the sort extraction. They can be viewed as control agents whose actions govern the interaction of the other agents and direct the entire process.

6.2 Relating experiential and formal views of heapsort

We can regard the definitive Heap model together with the agents introduced in the previous section (cf. Figure 6-4 and Box 6-2) as an ADM Heapsort artefact. Provided that we respect the standard patterns of interaction that are characteristic of heapsort, it can also be viewed as an ADM Heapsort device (cf. Chapter 5).

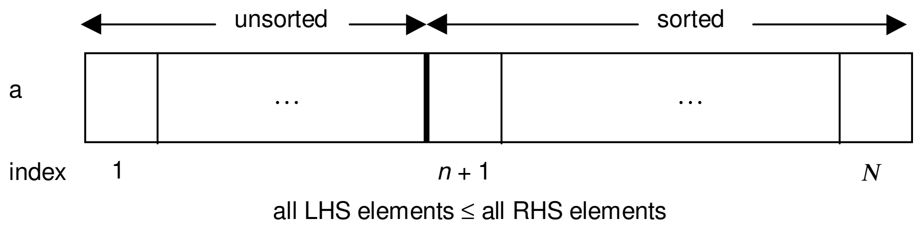
6.2.1 Integrating the Heap model with a formal specification

It is of interest to relate the non-standard development of the ADM Heapsort device described above to a more traditional approach to the development of a standard procedural heapsort program (or 'Heapsort program device' in the sense of Chapter 5). One such approach is the Weakest Precondition (WP) formal method, introduced by Dijkstra [Dij76], which can be used to derive a procedural program from a formal specification. An extract from a suitable formal specification for a heapsort program is shown in Box 6-3. The pseudo code for such program is given in Box 6-1. The extract in Box 6-3 is the specification of the sort extraction phase in the heapsort.

In Box 6-3, the key concepts in the specification are the pre- and post-conditions of the heapsort program, and the loop variant and invariant for the sort extraction. The post-condition describes the state we want the computer to have after the program (or piece of program) has been executed. The pre-condition describes the assumptions that the initial state must satisfy in order for the program (or piece of program) to operate correctly. Taken together, the loop variant and invariant guarantee termination and correctness of the sort extraction phase.

The formal specification for the heapsort process can be interpreted as specifying significant and characteristic states in the execution of any implementation of heapsort. In any implementation, there must be a point at which a heap is established on the array A (cf. the assertion of {heap(1, n)} in Box 6-3), and a point at which the largest element has been extracted etc. Because of the abstract program-independent nature of the formal specification, it refers to states and operations that can be interpreted with reference to the physical array and tree artefacts upon which the concept of heapsort is based (cf. Figure 6-1). The full specification for the heapsort process is included in Appendix B. All the variables and relationships in the formal specification of heapsort refer to observables that have commonsense interpretations in terms of these artefacts. This enables us to integrate the formal specification with the ADM Heapsort device.

PRE: $N \geq 1$
POST: $(\forall i,j \bullet 1 \leq i \leq j \leq N \Rightarrow a(i) \leq a(j)) \wedge a = perm(a_0)$
$heap(lo,hi) \Leftrightarrow (\forall i,j \bullet (lo \leq i \leq j \leq hi \wedge \exists k \bullet k > 0 \wedge i = j \; div \; 2^k) \Rightarrow a(i) \leq a(j))$



{**PRE**}
$n := N$;
"establish $heap(1,n)$"
{$heap(1,n)$}
do $n \neq 0 \rightarrow$ {Q}
"swap $a(1)$ and $a(n)$"
$n := n - 1$;
"re-establish $heap(1,n)$"
{Q}
od
{**POST**}

Loop variant is n.
Loop invariant is Q, defined:
$(0 \leq n \leq N) \wedge heap(1,n) \wedge$
$(\forall i,j \bullet n + 1 \leq i \leq j < N \Rightarrow a(i) \leq a(j)) \wedge$
$(\forall i,j \bullet 1 \leq i \leq n < j \leq N \Rightarrow a(i) \leq a(j))$

Box 6-3: A formal specification for the heapsort process cited from [BRS00]

The concept behind the extension of the Heapsort device is to include observation associated with the formal specification of heapsort. Such observation supplies an abstract trace of the heapsorting process as it might be inspected by a mathematician. The extended Heapsort device is depicted in Figure 6-5. The left-hand panel displays the current state of execution of a heapsort. The execution is in the heap establishment phase and the element *a[3]* is just about to be introduced. The right-hand panel takes on a different form according to which phase (Heap establishment or Sort extraction) of the heapsort algorithm is currently being inspected, and the values of invariants and variants are monitored as the algorithm is executed. The panel is divided vertically so that the left-hand part displays the relevant conditions, variants and invariants drawn from the formal specification in Box 6-3 and the right-hand part displays the current values of the associated variables and predicates.

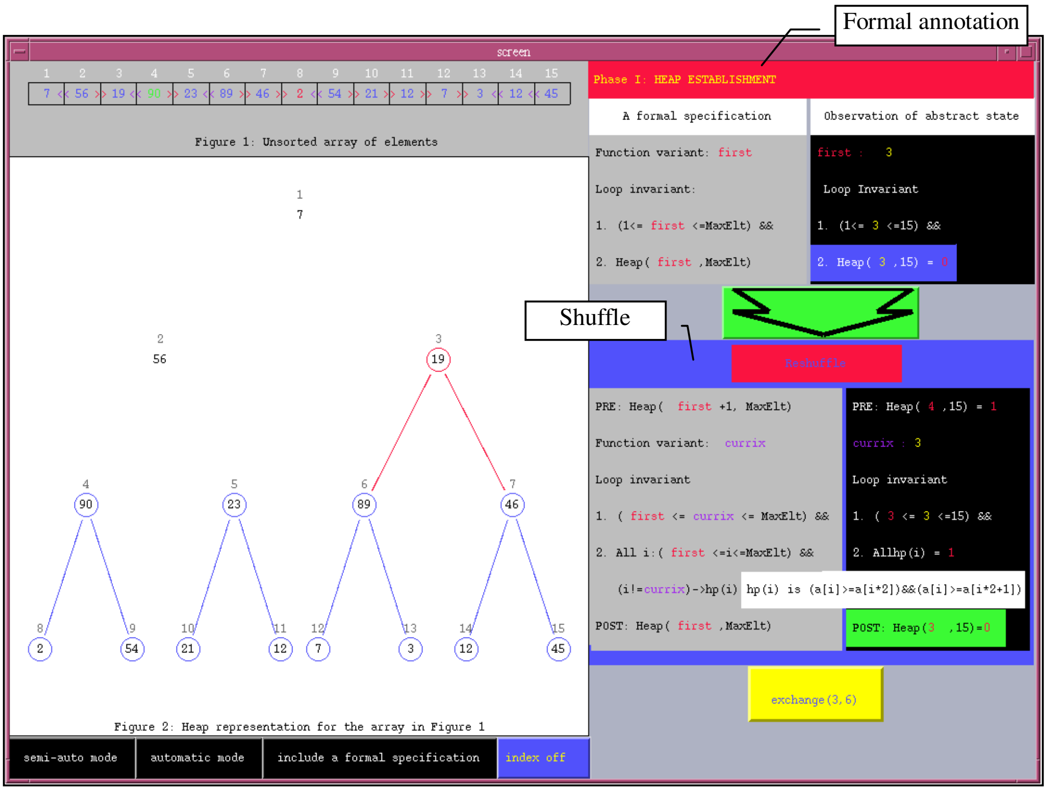


Figure 6-5: An extended version of heapsort with its formal specification

In the current state of the execution (with reference to the pseudo code in Box 6-1), the Shuffle procedure has just been invoked (with k = *first* = 3) in the heap establishment phase. In the right-hand panel of Figure 6-5, the top segment (respectively, the bottom segment) displays the current status of the loop variant and invariant for heap establishment (respectively, the Shuffle procedure).

In Figure 6-5, the variants and invariants of the specification are treated as observables in their own right, and are linked to the more primitive observables attached to the Heapsort device. Strictly speaking, this mode of observation of the Heapsort device is only appropriate in a restricted context for use, since it presumes that heapsorting activity is in progress, and makes references to observables concerned with control issues. For instance, each invariant is expressed as a predicate whose truth-value is dependent on the current state, as determined by the present status of both data and control. In practice, inspection of the current status of variants and invariants can be useful in interpreting more general interaction with the Heapsort device viewed as an ADM artefact (cf. Chapter 5).

6.2.2 The Heapsort device as a non-standard heapsort model

The Heapsort device together with its formal annotation, as depicted in Figure 6-5, can be used in a number of ways. The formal annotation serves as an instrument that monitors the state of the device in relation to its role in heapsorting, and gives additional insight into the interpretation of the observables associated with its control (e.g. first and last). As depicted in Figure 6-5, there are two possible modes of interaction. In the automatic mode, the user clicks on the Heap representation window and the heapsort proceeds automatically step-by-step, at each step performing the action which is displayed at the foot of the right panel. In the semi-automatic mode, the user takes over the roles of the agents heapmake and outsort in Box 6-2; only the agents maintainheap_i from Box 6-2 are present in the device, and they are adapted to respond to mouse clicks in the neighbourhood of the appropriate node. The possible applications for the Heapsort device with the formal annotation in normal use include:

- Learning about heapsort: the formal specification can be used to confirm that the heapsorting process is indeed being correctly followed;
- Learning about the formal specification: the heapsorting process may serve as a worked example for the purpose of confirming the accuracy of the formal specification.

As is the case with the ADM Jugs device studied in Chapter 5, many modes of interaction beyond normal use are possible with the ADM Heapsort device. For instance, the values of elements in an array can be changed while the heapsort is in progress. The impact of such changes depends upon the current status of the element reassigned and the nature of the value assigned to it. An analysis of the possibilities is useful in understanding the relationship between the ADM Heapsort device and a conventional heapsort program device and illustrating further potential applications for the Heapsort device:

- Scenario 1: *The assignment of a new value to an array element during the execution of heapsort does not disrupt the sorting process and respects the observables associated with the formal specification.*

  For instance, in the context of Figure 6-5, swapping *a[1]* and *a[2]* will not affect the outcome of the sort nor does it affect the validity of invariants.

- Scenario 2: *The assignment of a new value to an array element during the execution of heapsort temporarily disrupts the sorting process and invalidates pre-conditions and/or loop invariants associated with the formal specification*

For instance, in the context of Figure 6-5, swapping *a[4]* and *a[8]* will invalidate the pre-condition for the Shuffle procedure, but the Heapsort device will invoke the agents maintainheap3 and mantainheap4 (cf. Box 6-2) to restore the heap condition at nodes 3 and 4, after which the sorting will proceed normally.

- Scenario 3: *The assignment of a new value to an array element during the execution of heapsort disrupts the sorting process fatally and permanently violates predicates associated with the formal specification including the post-condition for the sort.*

  For instance, neither the traditional heapsort algorithm nor the Heapsort device sorts successfully if we swap the last two elements in the array after they have already been extracted from the heap.

Scenario 1 suggests ways of using the enhanced Heapsort device to explore the extent to which the traditional heapsort algorithm can cope with interference from its environment. The post-condition for the heapsort specification in Box 6-3 includes a clause (viz. *a = perm(a0)*) that prevents the arbitrary assignment of values to the array elements during execution. The standard heapsort algorithm will sometimes still operate successfully in a more general context where the values of array elements are subject to be reassigned during the execution but the post-condition only requires that they are correctly sorted on termination. Possible scenarios include, for instance, reassigning the largest element in the array to a larger value after it has been extracted or multiplying all the values in the array by a positive number.

Scenario 2 highlights the fact that the agents in the Heapsort device (cf. Box 6-2) and the program control in the Heapsort program device process the elements of the array differently. Without interference from the environment, the agents follow the standard pattern for maintaining the heap condition that is observed in the heapsort process, but they can also adapt to interference to a limited degree. For instance, they are always able to establish a heap despite any finite amount of interference in the heap establishment phase.

Scenario 3 suggests a natural way in which the observation of invariants and variants attached to the formal specification can be used to counteract the effects of random changes to the values to be sorted during the heapsort process. To illustrate this, we have created a variant of the Heapsort device in which such changes prompt the model to determine the optimal point to which the heapsort process has to be rewound. In this way, observation of the formal specification is used as a powerful form of meta-control that would normally entail intelligent action on the part of the user.

6.2.3 Construals of heapsort

The above discussion of the Heapsort device, and the related discussions in Chapter 5, highlight the significant distinction between MWDS and traditional techniques for programming and specification.

The enhanced ADM Heapsort device described in this chapter can be regarded a construal of heapsort developed using MWDS (cf. Figure 2-17). The three key ingredients in this construal are the definitive Heap model (cf. Figures 6-2 and 6-3), which is embedded in the underlying definitive script that determines the space for agent action, the agents and observations that are involved in performing the sorting activity (cf. Figure 6-4 and Box 6-2), and the particular patterns of action invocation associated with the heapsort process (cf. Boxes 6-3 and 6-1). The ADM Heapsort device offers cognitive support in understanding how all three ingredients work together subject to specific assumptions about the environment for interaction.

A crucial distinction between the ADM Heapsort device and more conventional animations of heapsort [GDL, 3DHPweb] lies in the physical nature of the ADM artefact and the open-ended quality of possible interaction with it. The function of construal can only be served by a model that can be tested beyond the limits of any preconceived and circumscribed range of interaction. If our formal specification is flawed, it is still important that it can be incorporated in the model. If the heapsort process is not correctly followed, there must be scope to reflect this deviation. More generally, a complete understanding of the heapsort process – if indeed there is such a thing – stems from insight into the way in which the process relies upon its context. In developing this insight, it is valuable – if not essential – to have scope for experimental interaction.

The characteristics of agents in the ADM heapsort device are 'continuous' observation and primitive actions (cf. the $\mathsf{maintainheap}_i$ agents in Box 6-2). In contrast, a procedural program such as heapsort acts as an agent that observes its input and thereafter acts upon its output by blindly following a sequence of actions selected from a complex pattern of possible actions according to the given input (cf. the Shuffle procedure in Box 6-1). In effect, the program is an agent that performs one large discrete observation and executes one of a large number of possible complex actions. The prior activity that is involved in conceiving such a program is essentially concerned with construal. The preconceived pattern of possible sequences of action that underlies a program (cf. the Shuffle procedure in Box 6-1) is derived by identifying very closely circumscribed regions within a space of agent interaction and anticipating all the necessary observation and action that will be required of each agent (cf. the $\mathsf{maintainheap}_i$ agents in Box 6-2). This typically demands strong assumptions about the stability of the environment, the reliability of actions and the predictability of their effects. MWDS, as depicted in Figure 2-17, can offer a framework for this construal.

The role of data structures and formal specification techniques in programs is to bring coherence to the intermediate states associated with its possible sequences of action. This involves identifying generic sequences of action that can be viewed as atomic (e.g. operations on a data structure) and characteristics of states that guarantee the validity of actions (e.g. pre-conditions in a formal specification). In combination, these techniques help to ensure that the state within the program is interpretable for purposes of analysis and that there is a correspondence between actions within the program and atomic operations on the data structure viewed as a physical artefact (cf. Box 6-3 and Figure 6-1). Other formal specification techniques characterise possible action sequences in more abstract terms, for instance, as traces (cf. the use of CSP in Figure 5-14) or in terms of state-transition patterns (cf. the use of a statechart in Figure 3-17).

In this thesis, we have been able to relate several different formalisms for specification and programming, viz. statecharts, CSP, WP and data structures, to MWDS. Each of these formalisms has been interpreted as relating to different types of agency, observation and patterns of actions that can be represented in an ADM artefact. This suggests that MWDS has the potential to address a number of topical themes in theoretical computer science:

- combining different formal and semi-formal approaches to system development to provide a more coherent and complete account of the system being constructed (cf. [HJ98], [AGT99]);
- relating formal methods and cognitive concerns (cf. [LV96], [Vinter98]);
- developing a broader foundational framework within which to study computation in the context of current and emerging computer applications.