leading to them. The occupied storage is not explicitly returned to the pool of available memory locations. It may then happen that the computer runs out of storage, while actually not all storage is being used. The task of a garbage collection algorithm is to return formerly used storage to the pool of available space.

In this chapter we aim at a general method for deriving garbage collection algorithms. The underlying idea is to start from a simpler "base algorithm" that is correct and well-understood, and to derive a concurrent garbage collection algorithm by applying transformations or by superimposing additional control. We will show that solutions to the termination detection problem can be almost mechanically transformed into solutions to the garbage collection problem. It turns out that virtually all existing (on-the-fly) garbage collection algorithms can be obtained by applying the transformation to a suitable termination detection protocol. Several new, highly parallel garbage collection algorithms can also be derived by following this approach.

In section 5.1 we introduce the garbage collection problem and present the transformations of Termination Detection algorithms to garbage collection algorithms. In section 5.2 we present a number of examples of the transformation. We conclude in section 5.3 with additional observations and remarks.

5.1 The Transformation

5.1.1 Garbage Collection

In many applications of computer systems the data is organized as a directed graph of varying structure. In this graph a fixed set of nodes exists, called the *roots*, which are the allowable entry points of the structure (for example, because these nodes are identified with the variables of the program). A node is called *reachable* if there is a directed path of edges leading from a root to the node. We refer to the subset of the reachable nodes as the *data structure*. Non-reachable nodes, i.e., nodes not belonging to the data structure, are called *garbage nodes*. A user program, also called the *mutator*, can add or delete edges between reachable nodes. The mutator never adds or deletes edges to or from garbage nodes. New nodes are allocated from a set of free nodes, called the *heap*, when new nodes are needed. The heap is mostly implemented as a linear list or a more general linked structure. We assume that there is a special root pointing to the heap, so heap nodes are

always reachable. Thus we can treat the "creation" and addition of a new node to the data structure as a sequence of mutations within the data structure. When an edge is deleted, a node may be disconnected from the data structure and become a garbage node. Garbage nodes cannot be made reachable again by the mutator, because no edges are added to garbage nodes.

We assume that the computer's memory is organized as an array of cells, each capable of representing one node of the directed graph. From now on we use the words cell and node interchangeably. A cell i can have several fields (representing the data), among which is a field *children*(i), containing the set of pointers to nodes to which an edge from i exists. Thus addition and deletion of edges consist of execution of the following code by the mutator:

ADD(i,j):
    { i and j are reachable nodes }
    children(i) := *children*$(i) \cup \{j\}$

DELETE(i,j):
    { i is reachable, $j \in$ *children*(i) }
    children(i) := *children*$(i) - \{j\}$

The task of a garbage collecting system is to identify garbage nodes and recycle them to the heap. Most garbage collectors are of the so-called "mark-and-sweep" type. Every round of such a collector consists of two phases. The first is the *marking phase*, which attempts to color the reachable nodes different from the garbage nodes. For this purpose an extra field *color* is added to each cell. This field can have the value *white* ("garbage") or *black* ("reachable"). (Later we introduce some more colors and extra fields.) An algorithm for the marking phase is also called a *graph marker*. The second phase is the *appending phase*, in which a sweep through the memory is made and the garbage nodes are appended to the heap. During the second phase the marking is undone, so the system is ready for the next round of garbage collection.

In this chapter we focus on the graph marking phase of garbage collecting systems. Marking algorithms do not really mark the garbage nodes, but rather mark the reachable nodes, starting from the roots. At the end of the marking phase the unmarked nodes are considered as garbage. Observe that the reverse is not always true: it is possible in some garbage collecting systems that garbage nodes have been marked, namely if they turned into garbage after they were visited by the graph marking algorithm. These nodes will consequently not be collected in the current round of the collector, but they will be in the next round. We define a collecting system to be *safe* if no reachable nodes are ever

appended to the heap.

Many algorithms for graph marking are based on a sequential traversal algorithm for directed graphs, like Schorr and Waite [SW67], or Wegbreit [We72]). These algorithms have the disadvantage that the mutator must be "frozen" during the marking phase. These garbage collectors are often called as an interrupt routine when the heap is (nearly) empty, and thus cannot be used in real-time applications. In the past ten years several algorithms were developed for *on-the-fly* garbage collection, in which the graph marking phase can be run concurrently with the mutator and yet it is guaranteed that all reachable nodes are being marked. See e.g. Dijkstra *et al.* [Dij78], Ben-Ari [Be84], Hudak and Keller [HK82], or Hughes [Hu85]. We study on-the-fly garbage collection in considerable detail in this chapter.

It is useful to distinguish several computational models for the on-the-fly garbage collection problem. In the early papers the problem was considered for the classical von Neumann type computer. There is one central processing unit, having access to one array of memory cells, and this processor runs the mutator program as well as the garbage collection program. Dijkstra *et al.* [Dij78] considered the possibility of using a second, special purpose, processor dedicated to garbage collection only. In this computational model there are two processors (or processes) working on the same data independently and concurrently. To minimize exclusion and synchronization overheads, the actions of the two processes are to interleave in as small a "grain" as possible. More recently, research has focussed on a more distributed type of computer system (see [HK82, Hu85]). The underlying motivation originates from the development of functional programming languages (LISP, SASL, etc). Functional programs are quite suitable for distributed evaluation and employ the kind of data structure we have defined. Here we assume a "pool" of processors, with each processor having its own array of memory cells. Of course, a child of a cell may now reside in the memory of another processor (i.e., links may be "interprocessor").

5.1.2 Graph Marking

In this section we develop the heuristics for transforming a termination detection protocol into an on-the-fly garbage collection algorithm. The graph marking algorithm will consist of a set of marking processes with a termination detection algorithm superimposed on it. The set of marking processes is always (almost) the same one, while many different termination detection algorithms are used. For this reason we refer to the superimposition as a transformation of the termination detection algorithm. For an extensive description and

treatment of the termination detection problem, see section 3.3 and chapter 4. It is assumed that each of a set of processes can be in an *active* or a *passive* state. A *passive* process can become *active* only by interaction with an *active* process. It can be shown that the state in which all processes are *passive* is stable. The purpose of a termination detection algorithm is to detect that the system is in this state.

We concentrate on the graph marking phase of garbage collection algorithms. Suppose that each cell in the memory contains a field *color*, and that initially *color*(*i*) = *white* for all *i*. The purpose of the marking phase is to blacken every node that is reachable from one of the root nodes. Then, the appending phase will collect the white nodes and whiten the black ones, so the collecting algorithm can be repeated. To avoid any reachable cells from being collected, we must ensure that all reachable cells are marked before the appending phase takes over. In deriving the essential theory, we first assume that the mutator is frozen while the graph marker is working, which means that the data structure is fixed. We subsequently adapt the mutator program so as to run concurrently with the marker safely.

5.1.2.1 The Basic Transformation. For each cell *i* in the computer's memory we introduce the (conceptual) process MARK1(*i*), defined as

```
MARK1(i):
    {wait until activated by external cause}
    forall j ∈ children(i) do
        if j was not activated before (* i.e., in this round *)
            then activate j ;
    color(i) := black ;
    stop. (* i.e., become passive *)
```

Let $M = \{0,1,...,N-1\}$ be the set of cells in the computer's memory and define the set of processes $\mathbb{P}$ by $\mathbb{P} = \{ \text{MARK1}(i) \mid i \in M \}$. In the following we sometimes refer to a cell as to its associated process and vice versa. Initially all cells are passive and white (we mean that all cells are white and all associated processes are passive). For an edge *e* from cell *p* to cell *q*, we say *p* is the *source* and *q* is the *target* of *e*.

Definition 5.1.2.1: *INV* is the property that for all edges, the source is active or white or the target is active or black.

Lemma 5.1.2.2: *INV* holds in the initial state of the system.

Proof: Obvious, because all cells are white in the initial state. □

All processes are waiting to be activated in the initial state. Of course, nothing happens unless processes are activated. The system is started by execution of the following code:

MARK_ROOTS:
forall $r \in$ *roots* **do** activate r.

The execution of MARK_ROOTS maintains *INV*. Some scheduling mechanism will be assumed or provided for the execution of the set of processes.

Lemma 5.1.2.3: *INV* remains true (while processes in $\mathbb{P}$ execute).

Proof: Initially *INV* holds as stated in lemma 5.1.2.2. The activation of a node by MARK_ROOTS trivially maintains *INV*. We proceed to show that *INV* is maintained by the execution of MARK1(i). To this end we remark first that after the activation of a cell the cell is active or black. Second, we use an additional invariant saying that before MARK1(i) arrives at the statement **stop**, all children of i have been activated.
No edges are added, and we consider the truth of *INV* with respect to an edge (i, j). *INV* can be violated only by (1) j becoming passive and white or (2) i becoming passive and black. But (1) transitions to passive white do not occur. Further (2), before MARK1(i) arrives at the statement **stop** it has activated all children of i. Hence we have our additional invariant that now all children have been activated. Thus, when i becomes passive and black, j is active or black. □

Lemmas 5.1.2.2 and 5.1.2.3 show that *INV* is an invariant of the system.

Lemma 5.1.2.4: Under the assumption that an active process eventually executes its next statement, the system will terminate.

Proof: $\mathbb{P}$ consists of only a finite number of processes. Each of them is activated at most once and runs to completion in finite time. □

Crucial for the correctness of our heuristic is the following lemma:

Lemma 5.1.2.5: Upon termination of the system, all reachable cells are black.

Proof: Upon termination all cells are passive and hence (because they have been activated) all roots are black. Upon termination all cells are passive and hence (by *INV*) edges with a black source have a black target. A cell is reachable if it is a root or there exists a path of edges from a root to the node. By induction it follows that the reachable cells are black. □

Lemma 5.1.2.6: Upon termination of the system, all black cells are reachable.

Proof: It is invariant that each active cell is reachable. Initially this holds. If a node is activated in MARK_ROOTS it is a root and thus reachable. If a node j is activated by MARK1(i) it is a child of i and by the invariant i is reachable, hence j is reachable. No edges are removed, so reachable cells remain reachable, which proves the invariant. Because only active cells are blackened, the result follows. □

Lemmas 5.1.2.2 through 5.1.2.5 prove the following theorem:

Theorem 5.1.2.7: When a termination detection protocol and a scheduling mechanism are superimposed on $\mathbb{P} = \{\text{MARK1}(i) \mid i \in M\}$, a correct graph marking algorithm is obtained.

By lemma 5.1.2.6 the graph marker marks only reachable nodes and hence the resulting collecting algorithm collects all garbage nodes in one round.

5.1.2.2 The Basic Transformation with Concurrent Mutator Actions.

Now assume the mutator is active concurrently with the graph marking system: edges can be added or deleted in an unpredictable way while the graph marking algorithm is executing. The correctness proof in section 5.1.2.1 is adapted accordingly.

We first study the effect of the deletion of edges. When edges are deleted, cells may become garbage during the marking phase even when they were marked already, and lemma 5.1.2.6 fails. (Its proof used the fact that no edges are removed.) So there can be garbage nodes that will not be collected. A weaker version of lemma 5.1.2.6 holds.

Lemma 5.1.2.8: Upon termination of the system, all black nodes were reachable at the beginning of the (current) marking phase.

Proof: It is invariant that each active cell was reachable at the beginning of the current marking phase. Initially this holds. If a node is activated in MARK_ROOTS it is a root and thus reachable. If a node j is activated by MARK1(i) it is a child of i and by the invariant, i was reachable at the beginning of the marking phase. If j was a child of i at the beginning of the marking phase it was also reachable, if it was added later then it was reachable at the time of addition, and hence also at the beginning of the marking phase. Even under deletion (and insertion) of edges reachability of a node *at a fixed moment in time* is stable. Because only active cells are blackened, the result follows. □

So a marked garbage node that remains uncollected is guaranteed to remain unmarked in the next marking phase. This means that any garbage node is guaranteed to be collected within two rounds of the collector. The DELETE primitive given in section 5.1.1.1

respects the graph marker's invariant *INV*. Thus, although deletions make the collector work "slower", they do not affect correct operation. Therefore deletions are allowed to take place concurrently with the graph marker algorithm.

A more serious problem is the addition of edges. The mutator may decide to add an edge between a passive black cell and a passive white one, thus violating *INV*. In this case the system as introduced in the previous subsection is incorrect. The following classical example, due to Dijkstra *et al.* [Dij78], shows that it is impossible to construct a safe graph marking algorithm without modifying the mutator program. Suppose a and b are reachable nodes and c is a node that is reachable only via a and b. Let the mutator enter the loop

repeat
DELETE (a,c) ; ADD (a,c) ;
DELETE (b,c) ; ADD (b,c)
until *false*.

Thus at any moment the data structure is in one of the three states shown in figure 5.1. Now assume the mutator always brings the graph into state 3 when the marking algorithm is inspecting a, and into state 2 when the marking algorithm is inspecting b. Then the collector never "sees" c and c will remain unmarked. The conclusion is that in order to obtain a correct graph marking system, it is necessary to put overheads on the mutator actions. We require the mutator to activate nodes also, in order to maintain *INV*. From now on addition of edges is done by (atomically) executing the following code:

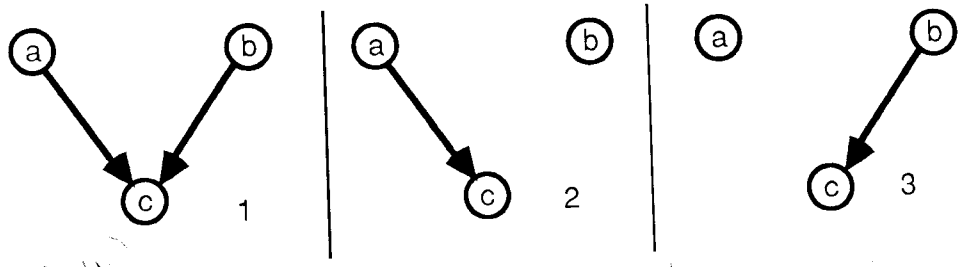

Figure 5.1

ADD(i,j):
(* i and j are reachable *)
children(i) := *children*(i) $\cup$ {j} ;
if i is not passive white **and** j is passive white **then** activate j.

Because the mutator touches reachable cells only, this modification of the mutator maintains the invariant in the proof of lemma 5.1.2.8.

Lemma 5.1.2.9: ADD(i,j) now maintains *INV* and *INV* is still an invariant of the system.

Proof: The new edge does not lead from a passive black to a passive white node because of the (possible) activation of j. The addition of the new edge respects *INV* for any edge leading to or from i or j or any other edge. Activation of j respects *INV* for edges leading to or from j. The status of all other edges remains unchanged. ADD(i,j) maintains the additional invariant in the proof of lemma 5.1.2.3, and it follows that also MARK_ROOTS and MARK1(i) still maintain *INV*. □

Stability of termination (see section 3.3.1.2) was proved under the assumption that only active processes could send activation messages. We must prove that this stability still holds under activations by the mutator.

Lemma 5.1.2.10: When the system has terminated, no activations by the mutator take place.

Proof: The mutator can activate a passive, white reachable node. When the system is terminated no such node exists, by lemma 5.1.2.5. □

It follows that the termination of $\mathbb{P}$ is stable. Also, termination detection protocols can work correctly under spontaneous activation by the mutator. Lemma 5.1.2.10 implies that when there is such an activation, there is at least one active node (or activation message) in the system. If the "spontaneous" activation is treated as an activation by this active node (or, by the sender of this message), the termination detection protocol will work correctly. As the proof of lemma 5.1.2.10 is not constructive, termination detection protocols that use acknowledgements and/or administration of messages may give difficulties. In these protocols it is necessary that an active node is explicitly found (see section 5.2.3).

The discussion results in the following main theorem:

Theorem 5.1.2.11: When a termination detection protocol and a scheduling mechanism are superimposed on $\mathbb{P}$ = {MARK1(i) | $i \in M$}, and the addition of edges is implemented as in ADD(i,j), a correct concurrent graph marker is obtained.

the path, starting at i and stepping from successor to successor, passes through every process exactly once and returns to i.

The termination detection algorithm is an example of our general construction of algorithms for Distributed Infimum Approximation; see section 4.2.5.3. It can be seen as a distributed version of the following code:

```
repeat
    success := true;
    forall processes p do
        begin
            wait until p is not active ;
            visit p
        end
until success.
```

Define an observation period of a process to be the time between two subsequent visits. Then "visit p" means:

if p was active since last visit **then** *success* := $false$.

Distributing this code is easy. The leader repeatedly sends out a boolean token on the ring, containing the value of *success*. To maintain the necessary information between two visits of the token, we assume there are two states for a passive process, namely *blue* and *idle*. (These states are *black* and *white*, respectively, in [DFG83]. We use other terms to avoid confusion with the marking colors.) Blue processes are passive processes that have been active earlier in their current observation period. Idle processes have been passive during their entire current observation period. So active processes that finish their job initially become blue. Only after the token passes does the process become idle. Active processes do not pass the token until they have turned blue.

The leader starts by sending out a *true* token. An idle process passes the token unchanged. A blue process passes the token as *false* to its successor and becomes idle. In this way a blue process reports the fact that it was active in the last observation period. When the leader recovers the token as *true* and is passive itself, termination is concluded. If a *false* token is recovered, a new (*true*) token is sent out.

Lemma 5.2.1.1: The DFG protocol satisfies the safety condition (see also theorem 4.1.3.2).

Proof: Assume the leader concludes termination. No process turned the token into *false*, so all processes were idle when they passed it. Hence all processes were passive during the entire last observation period. It follows from the way the observation periods are determined that there is a point in time where all processes were passive. This state is

stable, hence termination still holds. □

Lemma 5.2.1.2: The DFG protocol satisfies the liveness condition (see also theorem 4.1.3.2).

Proof: Suppose the system terminates before the token passes the leader for the i^{th} time. (That is, during the i^{th} tour of the token.) Then all processes are either blue or idle. During the $(i+1)^{th}$ tour at the latest all processes turn to idle, and at the end of the $(i+2)^{th}$ tour the leader concludes termination. □

In the original version of the DFG algorithm only the leader can conclude termination. The so-called *floating leader* variant is based on the following observation:

Lemma 5.2.1.3: When the token has visited all processes consecutively without encountering a blue process, the system is terminated.

Proof: Call the last process in this sequence the leader. Now the proof is as the proof of lemma 5.2.1.1. □

So instead of a boolean token we can use an integer token, which has the value k if it has visited k consecutive processes that were not blue when the token passed it. The only task of the original leader is to send a token <0> onto the ring to start the algorithm.

Remark: In this version of the floating leader variant it is necessary that the processes know the total number of processes in the ring. Another variant uses unique identifiers for the processes instead. A blue process that receives the token, stamps the token with its identity and passes it on. Termination is concluded by an idle process that receives the token with its own identity.

5.2.1.2 Derivation of the Graph Marking System. We will superimpose the floating leader variant of the DFG protocol onto the set of processes $\mathbb{P} = \{\text{MARK1}(i) \mid i \in M\}$, as outlined in section 5.1.2. Each cell can be in one of three states as far as the DFG protocol is concerned (active, blue or idle), and in one of two states as far as the marking is concerned (white or black). This yields a total of six states, which we represent by four colors according to table I.

Table I			
		marking colors	
		white	black
detector states	active	gray	"blue" (2)
	blue	.. (1)	blue
	idle	white	black

Remember that a process colors its associated cell black before turning passive. Hence combination (1) in table I never occurs. The statement "*color*(*i*) := *black*" in MARK1 is immediately followed by "**stop.**". State (2) occurs only between the execution of these two statements. Now, if we replace these statements by "*color*(*i*) := *blue*", we skip this state. (Note that the color of a node now indicates not only whether it has been marked or not, but also the state of its associated process.) The four remaining possible values of *color*(*i*) now have the following meaning:

white: node *i* is not marked and MARK1(*i*) is idle.

gray : MARK1(*i*) has been activated but has not run to completion yet. It is still active, and node *i* is not yet marked.

blue : node *i* is marked and MARK1(*i*) is a blue process.

black: node *i* is marked and MARK1(*i*) is a passive process.

In a gray node *i* the following transcription of MARK1 must be executed:

forall $j \in children(i)$ **do**
    if $color(j) = white$ **then** $color(j) := gray$;
$color(i) := blue$.

The leader starts the termination detection algorithm by sending a token <0> onto the ring. The token circulates, and is changed by the processes it passes as follows:

– White or black processes add 1 to the token value.

– Gray processes keep the token until they become blue, and then act as blue processes.

– Blue processes change the token into <0>, and change themselves to black.

A process that increases the value of the token to N concludes termination.

A natural choice for the successor of i is of course $S(i) = i+1 \ mod \ N$, and we can let the token start its journey in cell 0. We now combine the termination detection algorithm with the MARK1 processes, add a "scheduler" for the MARK1 processes, and

transform the resulting program to a complete graph marking algorithm. We do this in several steps. The first step is writing out the termination detection algorithm. The termination detection algorithm is simulated by the following program (*token* denotes the value of the token, *cell* the cell it is visiting):

```
(* Initiate token *)
token := 0 ; cell := N−1 ;
(* Circulate token *)
repeat
    (* Travel to next cell *)
    token := token+1 ; cell := (cell+1) mod N ;
    (* Wait, if cell is gray *)
    if color(cell) = gray then
        wait until color(cell) = blue ;
    (* Token becomes 0 if cell is blue *)
    if color(cell) = blue then
        begin token := 0 ; color(cell) := black end
until token = N.
```

In the second step we add the scheduling and execution of MARK1 processes. We must ensure that every gray MARK1 process eventually executes and become blue. We do this by substituting the code for MARK1 for the wait statement. Thus a gray process executes when it has the token; and only then. The procedure MARK_ROOTS must be executed before the termination detection procedure starts. This results in the following code:

```
(* MARK_ROOTS *)
forall r ∈ roots do color(r) := gray ;
token := 0 ; cell := N−1 ;
repeat
    token := token+1 ; cell := (cell+1) mod N ;
    if color(cell) = gray then
        begin (* execute MARK1(cell) *)
            forall j ∈ children(cell) do
                if color(j) = white then color(j) := gray ;
            color(cell) := blue
        end ;
    if color(cell) = blue then
        begin token := 0 ; color(cell) := black end
until token = N.
```

Note that all gray processes are eventually scheduled and hence the system is still guaranteed to terminate. The derivation of a graph marking algorithm is now complete. All components (marking processes and MARK_ROOTS, scheduling, and termination detection) are put together.

We apply a few simplifications to the resulting code. In the following transformation step we eliminate the color blue and combine the two **if**-statements to one. Observe that a node is blue only between the completion of MARK1 in that node and the assignment to *color* in the subsequent (second) **if**-statement. In fact, the blue color is used only to "trigger" the second **if**-statement in the main loop. Conversely, because processes can turn blue only as a result of the first **if**-statement, the second one is not executed if the first **if**-statement is not. Hence either the statements are both executed, or neither of them is. Thus the blue color can be eliminated by combining the two **if**-statements into one. This is done in the next version of the program, where also the "shorthand" *shade*(*j*) is introduced for "**if** *color*(*j*) = *white* **then** *color*(*j*) := *gray*":

```
(* MARK_ROOTS *)
forall r ∈ roots do shade(r) ;
token := 0 ; cell := N-1 ;
repeat
    token := token+1 ; cell := (cell+1) mod N ;
    if color(cell) = gray then
        begin forall j ∈ children(cell) do shade(j) ;
            token := 0 ; color(cell) := black
        end
until token = N.
```

(In fact, *shade*(*j*) is more than just a shorthand. The clause "**if** *color*(*j*) = *white* **then** *color*(*j*) := *gray*" can be implemented as the setting of a single bit. Encode *white* as 00, *gray* as 01 and *black* as 11, then it is equivalent to "set the second bit to 1". When one is interested in deriving a fine-grained system it is essential that the operation takes one access to *j*, not two.)

The reader is invited to compare this algorithm to the one given in Dijkstra *et al.* [Dij78] and note the similarities. In [Dij78] it is not observed that it is possible to eliminate one arithmetic operation ("*token* := *token*+1") from the loop. Instead of using a token with a counter (the first variant of the floating leader DFG we presented) we can use a token with the identity of the last process that received the token when it was gray (see the remark at the end of section 5.2.1.1). The resulting on-the-fly garbage collection algorithm is

```
forall r ∈ roots do shade(r) ;
id := 0 ; cell := 0 ;
repeat
    if color(cell) = gray then
        begin forall j ∈ children(cell) do shade(j) ;
            id := cell ; color(cell) := black
        end ;
    cell := (cell+1) mod N
until cell = id.
```

5.2.1.3 Concurrent Mutator Activities. According to section 5.1.2.2, the graph marker designed in section 5.2.1.2 allows concurrent mutator modifications in the data-structure. The mutator executes the following code indivisibly when it adds an edge (*i*,*j*):

```
ADD(i,j):
    children(i) := children(i) + {j} ;
    shade(j).
```

Because the DELETE action preserves the invariants of the system it can remain unchanged, as argued in section 5.1.2.2. Because we assumed (section 5.1.1) that the heap nodes are reachable nodes, the "extension" of the data structure with new nodes is not a new type of mutation, but rather a series of additions and deletions of edges between reachable nodes. This concludes the derivation of the essential phase of the on-the-fly garbage collection algorithm of Dijkstra *et al.* [Dij78].

5.2.2 A Highly Parallel Garbage Collector

As in section 5.2.1 we assume here that the mutator and collector processors share one array of memory cells. In this section we introduce a highly parallel garbage collector, i.e., a collector that consists of many garbage collecting processes. The collector and its underlying termination detection protocol are generalizations of those in section 5.2.1.

5.2.2.1 A Highly Parallel Termination Detection Protocol. The DFG protocol basically consists of sequentially visiting all processes. The protocol is described by the following code, where the passing of the token ensures that the inner loop is in fact executed sequentially.

```
DFG:
    repeat
        success := true ;
        forall processes p do
            begin wait until p is not active ;
                if p is blue then
                    begin success := false ; p becomes idle end
            end
    until success.
```

(This is the original version, not the floating leader variant.) The fact that the DFG protocol executes the inner loop sequentially is not essential, and also is not used in its correctness proof. Hence any protocol is correct in which all processes are visited exactly once during each iteration of the main loop. An extensive study of the structure of termination

detection algorithms is found in chapter 4. The basis of the on-the-fly garbage collection algorithms in this section is the following termination detection protocol skeleton:

repeat
    success := *true* ;
    forall *i* **do** "visit *i*"
until *success*.

Here "visit *i*" consists of a termination detection visit (as in section 5.2.1) as well as a possible execution of MARK1.

5.2.2.2 A Highly Parallel Graph Marking System. In order to turn the protocol skeleton for distributed termination detection into a graph marking system, three things need to be done:

- Add the code for MARK_ROOTS,
- Specify the code for "visit *i*", and
- Supply a scheme according to which nodes are visited.

Using the same color and notations as in section 5.2.1, the code for MARK_ROOTS is of course

MARK_ROOTS:
    forall $i \in roots$ **do** *shade*(*r*).

A "visit" has the same semantics as in section 5.2.1. Again we combine the termination detection protocol with a scheduler, to execute the code for MARK1 in gray nodes. So *visit*(*i*) becomes the following routine:

VISIT(*i*):
    if *i* is *gray* **then** MARK1(*i*) ;
    "termination detection visit to node *i*".

or, if we combine the two statements into one as we did at the end of section 5.2.1.2

VISIT(*i*):
    if *color(i)* = *gray* **then**
        begin forall $j \in children(i)$ **do** *shade*(*j*) ;
            color(*i*) := *black* ; *success* := *false*
        end.

We supply two parallel visiting schemes to complete the garbage collecting systems. The first scheme works for an arbitrary number of marking processors, the second scheme works for exactly two marking processors. For the first scheme, assume that there are *k*

processors available for garbage collection. The simplest traversal scheme for the processes is to partition the set M of cells into k parts, and assign each garbage collection processor to a part. So let $\{ S_i \mid 1 \leq i \leq k \}$ be a partition of M. The marking system is given by:

```
forall r ∈ roots do shade(r) ;
repeat
    success := true ;
    forall i ∈ {1,..,k} pardo
        forall p ∈ S_i do visit(p)
until success.
```

See Cohen [Co85] for a more detailed garbage collection algorithm along these lines. This visiting scheme has the disadvantage that the partition of the memory cells must be fixed in advance. It is not possible to adapt the workload of a processor to its speed dynamically. This is possible in our second scheme, suited for two processors. Let the two processors start at opposite ends of the cell array, and work towards each other. When they meet somewhere, all processes have been visited. Call the two garbage collecting processes GCA and GCB. We can describe the two processes as follows:

GCA:

```
forall r ∈ roots do shade(r) ;
repeat
    synchronize ;
    success := true ;
    a := 0 ;
    synchronize ;
    repeat visit(a) ;
        a := a+1
    until a > b ;
    synchronize
until success.
```

GCB:

```
repeat
    synchronize ;

    b := N−1 ;
    synchronize ;
    repeat visit(b) ;
        b := b−1
    until b < a ;
    synchronize
until success.
```

Here the statement **synchronize** is a synchronization primitive: it is assumed that a processor that comes to this statements waits until the other processor also comes to a statement **synchronize**. Then they pass this point in the program simultaneously. The reader is invited to improve on this algorithm in two ways: (1) decrease the synchronization overheads, and (2) make a dynamic scheme for more than two processors.

5.2.2.3 Concurrent Mutator Actions. Concurrent mutator actions in these graph marking systems are handled as in section 5.2.1.3.

5.2.3 The Marking System of Hudak and Keller

Hudak and Keller [HK82] presented a garbage collector that is suitable for a completely distributed environment. An arbitrary number of mutator and collector processes can be active at any time. In this section we show that their algorithm is obtained by superimposing the distributed termination detection protocol of Dijkstra and Scholten [DS80] on the set $\mathbb{P}_2$ of processes MARK2(i). We discuss the Dijkstra and Scholten protocol, its transformation to the Hudak and Keller algorithm according to section 5.1.2, how concurrent mutator actions can be allowed using this algorithm, and how more concurrent mutator actions can be supported.

5.2.3.1 The Distributed Termination Detection Protocol of Dijkstra and Scholten [DS80]. Basically the protocol by Dijkstra and Scholten (hereafter called the DS protocol) is an acknowledgement scheme for activation messages. It is assumed that the initial source of all activity in the network is one special process E. Each process p keeps two counters:

$C(p)$ = the number of activation messages that p has received but not yet acknowledged, and

$D(p)$ = the number of activation messages that p has sent but not yet received an acknowledgement for.

We call a process p *engaged* iff $D(p)>0$ **or** $C(p)>0$. For an engaged process p, its *engagement message* is the message that caused it to become engaged, and p's *activator* or *father* is the sender of p's engagement message. $C(E) = 0$ always and E has no father. p is required to acknowledge all activation messages it receives (this action is called *signaling* in [DS80]), but it is not allowed to acknowledge its engagement message as long as it is active or $D(p)>0$. As soon as p is passive and $D(p)= 0$, p is assumed to acknowledge all messages in finite time, its engagement message as the last. Dijkstra and Scholten prove the following facts for this signaling scheme:

Lemma 5.2.3.1: (Safety) When E becomes unengaged, the system is terminated.

Lemma 5.2.3.2: (Liveness) When the system is terminated, E becomes unengaged within finite time.

The algorithm is described as a very general, non-deterministic scheme (p is free to decide when it signals the other messages it receives). It is enough for p to maintain the number of ACKs it still has to receive ($D(p)$), rather than keep a set of unacknowledged messages. Also, when p acknowledges a message, p need not mention the message concerned.

5.2.3.2 Derivation of the Graph Marking System. For the purpose of deriving the graph marking system we assume that each message, except the engagement message, is acknowledged immediately. Hence the value of $C(i)$ can only be 0 (for an unengaged process) or 1 (for an engaged process). When running MARK2(i), a test for earlier activations is necessary anyhow. By the primitive *activate*$(i, father)$ we mean: send an activation message, containing the sender's identity *father*, to i. The receipt of this message by i triggers execution of the procedure ACTIVATE($i, father$), which contains the code for MARK2(i) as well as the code for the DS protocol:

```
ACTIVATE(i, father):
    if i was not activated before then
        begin C(i) := 1 ;
            forall j ∈ children(i) do
                begin activate(j,i) ; D(i) := D(i)+1 end ;
            color(i) := black ;
            while D(i)>0 do
                begin receive an ACK ; D(i) := D(i)− 1 end ;
            signal(father) ; C(i) := 0
        end
    else (* i.e., i is or has been engaged already *)
        signal(father).
```

The primitive *signal*$(father)$ means: send an acknowledgement to the node *father*.

The complete graph marking algorithm that we derive now is "message driven", i.e., something can happen only upon the receipt of a certain message. We write the algorithm in "message driven form", i.e., with a piece of code for every possible message that can arrive. This piece of code is run to completion before the next message is accepted, thus ensuring atomicity of the described actions. State information about the process is stored explicitly. Suppose node i contains yet another field $father(i)$. We present the message driven form. Again, *signal*$(father(i))$ means that a signal is sent to $father(i)$. The receipt of such an acknowledgement by i triggers execution of SIGNAL(i).

```
ACTIVATE(i, father): (* executed if i receives an activation message from father *)
    if i was not activated before then
        begin father(i) := father ; C(i) := 1 ;
            forall j ∈ children(i) do
                begin activate(j,i) ; D(i) := D(i)+1 end ;
            if D(i)= 0 then (* i has no children *)
                begin color(i) := black ; C(i) := 0 ;
                    signal(father(i)) end
        end
    else (* i.e., i was activated before *)
        signal(father).

SIGNAL(i): (* executed if i receives an acknowledgement signal *)
    D(i) := D(i)- 1 ;
    if D(i)= 0 then
        begin color(i) := black ; C(i) := 0 ; signal(father(i)) end.
```

Note that we have deferred "*color*(*i*) := *black*" to the time of unengagement of *i*. This minor change does not affect the correctness or termination properties of the algorithm. We do this to ensure that the statements "*color*(*i*) := *black*" and "*C*(*i*) := 0" always appear together. Soon we replace these two statements by one, similarly to what we did in section 5.2.1.2. Except between the two statements mentioned, a node is always in one of the following three situations.

State 1: $C(i) = 0$, *color*(*i*) = *white*,
State 2: $C(i) = 1$, *color*(*i*) = *white*,
State 3: $C(i) = 0$, *color*(*i*) = *black*.

As in section 5.2.1 we use a coding trick and represent *C* in the *color* field by using the extra color *gray* to represent state 2. The test "*i* was not activated before" is then replaced by "*i* is *white*". The program for all node processes now becomes

```
ACTIVATE(i, father):
    if color(i) = white then
        begin father(i) := father ; color(i) := gray ;
            forall j ∈ children(i) do
                begin activate(j,i) ; D(i) := D(i)+1 end ;
            if D(i)= 0 then
                begin color(i) := black ; signal(father(i)) end
        end
    else
        signal(father).

SIGNAL(i):
    D(i) := D(i)− 1 ;
    if D(i) = 0 then
        begin color(i) := black ; signal(father(i)) end.
```

The marking process is elegantly started and controlled by the following transcription of MARK_ROOTS:

```
E:
    (* MARK_ROOTS *)
    forall r ∈ roots do
        begin activate(r,E) ; D(E) := D(E)+1 end ;
    while D(E) > 0 do
        begin receive an ack ; D(E) := D(E)− 1 end.
```

Upon termination of E the marking has completed. Of course this part of the algorithm can be written in message driven form also. This part is somewhat underdeveloped in [HK82]. The original algorithm was suited only for graphs with one root.

We have ignored the scheduling of the processes. It is assumed that the run-time system ensures that messages are eventually received and the procedures above are executed. The algorithm can be run on an arbitrary number of processors, each with its own local memory, and allows an arbitrarily large number of mutator processes to run concurrently with it (under the restrictions derived in the next subsection). However, exclusive access to cells is necessary. In [HK82] suitable "locking machinery" is built in to guarantee exclusive access. This machinery is also ignored here.

5.2.3.3 Concurrent Mutator Activities. Some difficulties must be overcome when we want to use the graph marker of the preceding subsection as a concurrent graph marker.

We now study how the graph marker of [HK82] handles it. The invariant *INVa*, as defined in section 5.1.2, has the following form for the algorithm of Hudak and Keller:

(A) If i is a gray node then activation messages have been sent to all of its children,

(B) If i is a black node then no child of i is white.

We have B because i turns black only after receiving acknowledgements for the activation messages it sent. The mutator must respect this invariant and thus it must activate nodes sometimes. But, in order to enable a node to send an acknowledgement later, it must be given a father. Hence the mutator must be able to find a gray node that is willing to "adopt" the node. And, although lemma 5.1.2.10 guarantees that such a node exists, the problem remains to find a suitable one. This is why in [HK82] the behavior of the mutator is limited where the addition of edges is concerned. Only in a small number of specific cases may edges be added.

One of these cases and its solution (see [HK82]) is the following: the addition of an edge (a,c) by a primitive *add_grandson*(a,b,c), where it is assumed that edges (a,b) and (b,c) exist already. Obviously *add_grandson*(a,b,c) does not violate the invariant when a is white (there are no requirements on c in that case) or b is black (c is non-white already). This is also the case when both a and b are gray. By B it is impossible that a is black and b is white. The remaining cases are (1) and (2) in table II.

Table II

		a		
		white	gray	black
b	white	-	(1)	impossible
	gray	-	-	(2)
	black	-	-	-

Case 1: Suppose a is gray and b is white. Then an activation message has been sent to b, but not necessarily to c. In order to maintain invariant A, we have to send c an activation message and here we can make a c's father, i.e., a adopts c.

Case 2: Suppose a is black and b is gray. In this case we choose b to adopt c. Before we can make the link, we must wait until the activation has resulted in c becoming gray, for according to B, it is not enough that an activation message is on its way to c.

We can now give the primitive ADD_GRANDSON.

```
ADD_GRANDSON(a,b,c)
    (* a is reachable, b ∈ children(a), and c ∈ children(b) *)
    if color(a) = gray and color(b) = white then
        begin activate(c,a) ; D(a) := D(a)+1 end ;
    if color(a) = black and color(b) = gray then
        begin activate(c,b) ; D(b) := D(b)+1 ;
            wait until ACTIVATE(c,b) is finished in c
        end ;
    children(a) := children(a) + {c}.
```

For more primitives that can be supported in a similar way, see [HK82, Hu83].

5.2.3.4 Allowing more Concurrent Mutator Activities. With some effort it is possible to implement a more general ADD operation under the Hudak/Keller garbage collector. Suppose the mutator adds an arbitrary edge (a,c). Overhead is not necessary in all cases that can occur. If a is still white, or c is gray or black, nothing needs to be done. The need for cooperation in implementing a general ADD operation is summarized in table III:

Table III

		a		
		white	gray	black
c	white	-	(1)	(2)
	gray	-	-	-
	black	-	-	-

In two cases (see table III) special action is needed:

Case 1: If a is gray and c is white, we can maintain the invariant by having a adopt c.

Case 2: If a is black already, this is impossible but c must be at least gray before the link can be made. The solution is to find an arbitrary gray node b and force it to adopt c. That is, we send c an activation message bearing b as sender. When this message is processed by c, c is (at least) gray and the link can be made.

The ADD operation can be implemented by the following program:

```
ADD(a,c):
    if color(a)=gray and color(c)=white then
        begin activate(c,a) ; D(a) := D(a)+1 end ;
    if color(a)=black and color(c)=white then
        begin b := ... ; (* any gray process; see the discussion below *)
            activate(c,b) ; D(b) := D(b)+1 ;
            wait until ACTIVATE(c,b) is finished in c
        end ;
    children(a) := children(a) + c.
```

The key problem of course is finding a suitable gray node b as discussed in case 2. We know by lemmas 5.1.2.10 and 5.2.3.1 that there is one. We give several suggestions to find a suitable process b.

(1) The root process E is always engaged, as long as there is any engaged node (lemma 5.2.3.1). So one can take $b = E$ always. This solution has some important disadvantages.
   - In most cases E will not reside on the same processor as a and/or c, hence this choice can increase communication complexity considerably.
   - When there are many ADD operations E will become a bottleneck. E will be blocked most of the time, and the processor where it resides will be busy most of the time handling ADD operations from other processors.

(2) Each processor can keep a pool of cells that are currently gray in its memory. This pool is updated by the activate and signal procedures. When a gray cell is needed, the host of a and/or c check their pools for gray cells. If there is no gray cell in that processor, they can ask their neighbors, etc.

(3) Another idea would be to add some "special purpose" nodes to the data structure. Suppose there is one special root S_p on each processor p. S_p has no "natural" children, but will adopt nodes when such is necessary. S_p is a root and will be grayed immediately after the start of the marking phase. An extra mechanism must be built in to ensure that
   - S_p will remain engaged as long as the marking phase goes on, so it will be available when necessary, and
   - S_p will become unengaged when the marking phase is finished, so it will not unnecessarily block the garbage collecting process.

   This implies that we must superimpose yet another termination detection protocol on the system, which means that this method is not feasible.