THE DERIVATION OF GRAPH MARKING ALGORITHMS

FROM DISTRIBUTED TERMINATION DETECTION PROTOCOLS

G. Tel, R.B. Tan, J. van Leeuwen

RUU-CS-86-11
August 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

THE DERIVATION OF GRAPH MARKING ALGORITHMS FROM DISTRIBUTED TERMINATION DETECTION PROTOCOLS

G. Tel, R.B. Tan, J. van Leeuwen

Technical Report RUU–CS–86–11
August 1986

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
The Netherlands.

THE DERIVATION OF GRAPH MARKING ALGORITHMS FROM DISTRIBUTED TERMINATION DETECTION PROTOCOLS[1]

Gerard Tel[2], Richard B. Tan[3] and Jan van Leeuwen.

Department of Computer Science, University of Utrecht,
P.O. Box 80.012, 3508 TA Utrecht, The Netherlands.

0. Abstract: We show that on-the-fly garbage collection algorithms can be obtained by transforming distributed termination detection protocols. Virtually all known on-the-fly garbage collecting algorithms are obtained by applying the transformation. The approach leads to a novel and insightful derivation of e.g. the concurrent garbage collection algorithms of Dijkstra et. al. [10] and of Hudak and Keller [12]. The approach also leads to several new, highly parallel algorithms for concurrent garbage collection. We also analyse a garbage collecting system due to Hughes from our current perspective.

1. Introduction.

In the past 10 years several algorithms for so-called "on-the-fly" garbage collection have been developed, e.g. [3,10,12,14,15]. Early papers concentrated on solutions that are as "fine-grained" as possible, i.e. solutions that allow a high degree of interleaving of the different processes at work. However, these papers offer no methodology for the construction of the various algorithms. Other papers give a detailed overview of an algorithm without trying to make it finer grained, like e.g. [12,14,15]. It seems that each on-the-fly garbage collection algorithm has its own privat "ad-hoc" idea and there is hardly any common background in these algorithms.

---

(1) This work was carried out while the second author visited the University of Utrecht, supported by a grant from the Netherlands Organization for the Advancement of Pure Research (ZWO).

(2) The work of this author was supported by the Foundation for Computer Science (SION) of the Netherlands Organisation for the Advancement of Pure Research (ZWO).

(3) Author's address: Dept. of Mathematics and Computer Science, University of Sciences and Arts of Oklahoma, Chickasha, OK 73018, USA.

In this paper we aim at a general methodology for deriving on-the-fly 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 transformation or by superimposing additional control. According to Dijkstra [7], C.S. Scholten first noticed the analogy between the problem of concurrent graph marking (the most difficult part of on-the-fly garbage collection) and a problem in the field of distributed computation known as the distributed termination detection problem [11]. We will show that solutions to the latter problem can be almost mechanically transformed into solutions to the former. It turns out that virtually all existing algorithms can be obtained by applying the transformation to a suitable termination detection protocol. Several new, highly parallel garbage collecting algorithms can also be derived by following this approach.

The paper is organized as follows. In the remainder of this section we give a brief introduction to the on-the-fly garbage collection problem (see e.g. Cohen [6] for a more extensive treatment) and the distributed termination detection problem (see e.g. Beilken et. al. [2] for a more extensive treatment). In section 2 we present the heuristics we use to transform distributed termination detection protocols into graph marking algorithms. In section 3 we apply the transformation to the termination detection protocol of Dijkstra, Feijen and van Gasteren [8] and demonstrate that the graph marking phase of the garbage collector due to Dijkstra et. al. [10] is obtained in this way. In section 4 we do the same with the termination detection protocol of Dijkstra and Scholten [9] to obtain the graph marking algorithm of Hudak and Keller [12]. In sections 5 and 6 we present several new, highly parallel graph marking algorithms, based on the distributed termination detection protocols by Tan and van Leeuwen [20] and Dijkstra, Feijen and van Gasteren [8]. In section 7 we outline a two-level "hierarchical" termination detection protocol and develop the corresponding graph marking algorithm. In the same section we will point out that the on-the-fly garbage collection algorithm presented by Hughes [14] is an intricate refinement of this algorithm. In section 8 we offer some final comments.

1.1. On-the-fly 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. A node is called *reachable* if it is reachable from at least one root via a path of edges. 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 edges to garbage nodes, but allocates new nodes from a list of free nodes called the *heap* when new nodes are needed. We assume that there is a special root pointing to the heap, so heap nodes are always reachable. When an edge is deleted, a

node may be disconnected from the data structure and become a garbage node. Garbage nodes can not be made reachable again by the mutator.

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 will 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.

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 will introduce some more colors and extra fields.) An algorithm for the marking phase will also be 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 that the system is ready for the next round of garbage collection.

In this paper 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 will be possible in some on-the-fly 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 a 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 traversal algorithm for directed graphs (see e.g. Schorr and Waite [18] or Wegbreit [22]). These (sequential) algorithms have the disadvantage that the mutator must be "frozen" during the marking phase. These garbage collectors are often called as an interrupt when the heap is (nearly) empty and can not 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 in parallel with the mutator and yet guarantee that all reachable nodes are being marked. See e.g. Dijkstra et. al [10], Ben-Ari [3], Hudak and Keller [12] or Hughes [14]. We will be studying on-the-fly garbage collection in considerable detail in this paper.

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 (cf fig. 1). 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. [10] considered the possibility of using a second, special purpose,

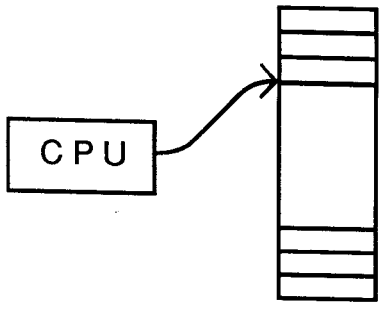

Figure 1

processor dedicated to garbage collection only (see fig. 2). In this computational model there are two processors working on the same data in an "independent" manner. In this model we want to allow the actions of the two processes to interleave in as small a "grain" as possible, to minimize exclusion and synchronization overhead. More recently, research has focussed on a more distributed type of computer system (see fig. 3, cf [12,14]). 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 defined. Hence it is natural to think of 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").

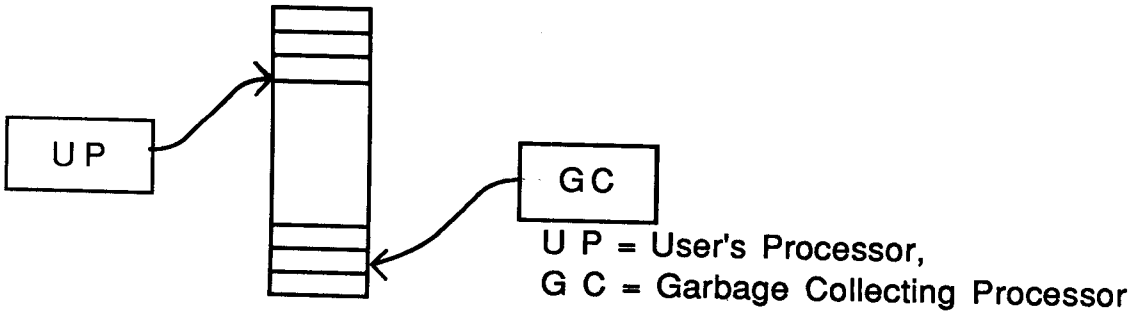

Figure 2

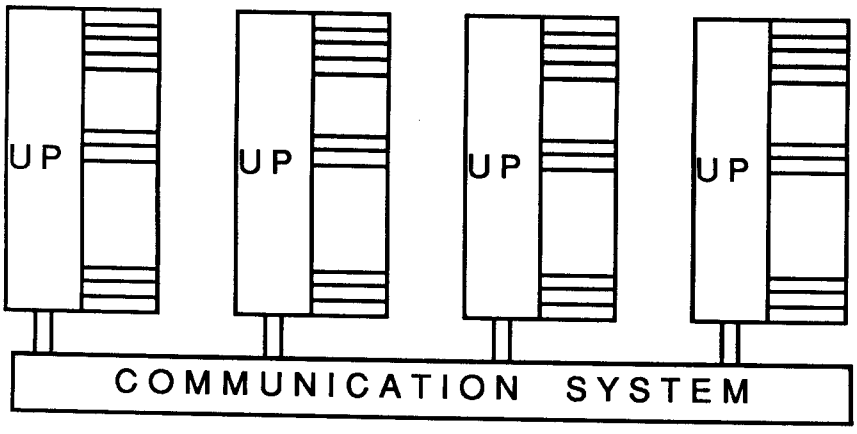

Figure 3

1.2. Distributed termination detection.

Distributed termination detection is the problem to determine when all activity in a distributed system has ceased. Assume each member of a set of processors is performing a certain task, which it will eventually finish. During its work, a processor may decide to send new tasks to other processors. It is possible that in this way a processor is "woken up" again, after having been idle for a while. It is usually assumed that no new task is created in an idle processor after the initialization of the system and that only active processors can wake up idle ones. Clearly, when all processors have finished their current activities, and no messages are in transit anymore, the system will have entered a stable state in which all processes are idle.

A more formal definition of the distributed termination detection problem is as follows. Let $\mathbb{P}$ be a set of processes, each of which can be in one of two states, namely *active* or *passive*. (The passive state will also be called *idle*.) Only active processes may send so-called *activation messages* to other processes. A process may change its state, with the restriction that a change from passive to active may take place only upon receipt of an activation message. We assume that a process can change from active to passive any time it wants to. We say the system $\mathbb{P}$ is *terminated* when all processes in $\mathbb{P}$ are in the passive state and there are no activation messages under way. Again it is obvious that this state is stable: when terminated, $\mathbb{P}$ remains terminated forever. Termination detection can now be formulated as the problem to determine that this state is reached, and a *termination detection protocol* is an algorithm that can be superimposed on the processes in $\mathbb{P}$ to enable them to do so. A good termination detection protocol must satisfy the following criteria (see Apt, [1]):

(1) Safety: no termination is detected unless there really is termination,

(2) Liveness: if there is termination it will be detected.

Many termination detection protocols are reviewed in [2,4]. As we will be using termination detection protocols extensively in this paper, we give the idea behind some of them. Suppose for a while that sending and receiving of a message is instantaneous. We illustrate the behaviour of the processes in $\mathbb{P}$ by time diagrams (cf fig. 4), in which each horizontal line represents the behaviour of one process in $\mathbb{P}$. The horizontal axis represents time. By a "blocked" interval (—▬▬—) we indicate that a process is active for the duration of the interval and by an arrow (↑,↓) we mean the exchange of an activation message.

If every process would send a report about its state at a certain time t to some central site, these reports would enable one to determine the state of the system at time t. For example, in figure 4, at t_1 some processes report an active state and hence the system is not terminated. At t_2 however, all processes report a passive state and hence termination can be concluded. The protocol based on this idea would require synchronized clocks (i.e. global time), see Rana [17]. In systems that do not support synchronized local clocks this simple protocol is unsafe, as the following scenario shows (see fig. 5): process p_1's clock is fast and p_1 reports its passive state somewhat before t. Then p_2 activates p_1 and becomes passive. Now the clocks at p_2 and p_3 read t and they report their state as passive. Although the system is not terminated, all processes reported a passive state and termination is (erroneously) concluded.

It turns out that we must observe each process during a certain interval, rather than at a single point in time. The following theorem is due to Chandy and Misra [4]:

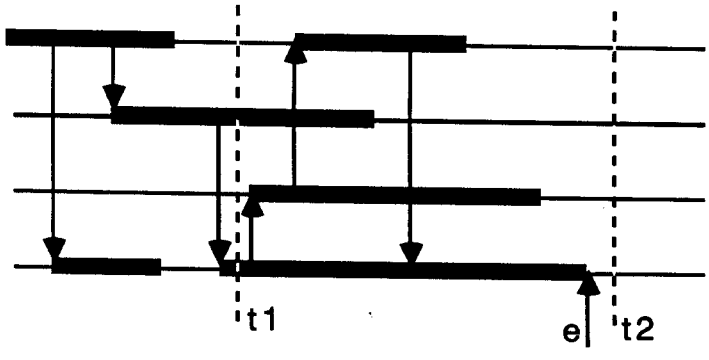

Figure 4

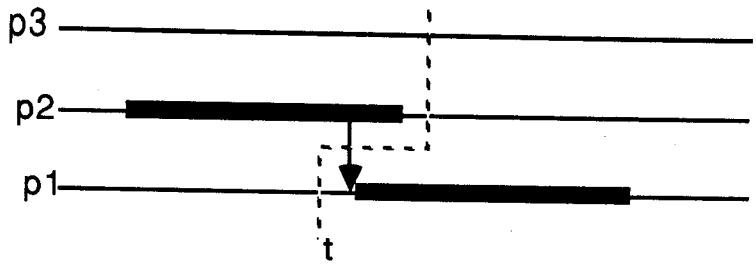

Figure 5

Theorem 1.1: Observe each process p_i during an *observation interval* $(start_i, end_i)$ and ensure that for all i, j: $start_i < end_j$ and that no p_i has an unprocessed message on any incoming link at time $start_i$. If no p_i was active during its observation interval, then the system is terminated.

Proof: There is a time t that is contained in each observation interval. Each process was passive at t and there were no messages under way. Hence the system was terminated at t, and still is. □

Theorem 1.1 will enable us to give correctness proofs of most of the termination detection protocols we use in this paper, like the protocols due to Dijkstra, Feijen and van Gasteren [8] and Tan and van Leeuwen [20] and the variants we use of these protocols.

It is possible to adapt the protocols (and theorem 1.1) to the more realistic situation that activation messages actually take some time to reach their destination. This is not done here, but it is for example in Tel [21].

2. Graph marking.

In this section we will develop the heuristics for transforming a termination detection protocol into an on-the-fly garbage collection algorithm. We will concentrate on the graph marking phase. 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 color every node black 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 will first assume that there is no concurrent mutator activity, meaning that the data structure is fixed. We will subsequently adapt the mutator program so as to run concurrently with the marker safely.

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}
    for all 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 $M = \{1,2,...,M-1\}$ 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 will identify a cell and its associated process. We will speak of "white processes", "active cells", etc. Initially all cells are passive and white.

Definition: *INV* is the property that no passive black cell has a passive white child.

Lemma 2.1: *INV* holds when the system is not activated.

Proof: Obvious, because there are no black cells. □

Of course, all processes keep waiting until they are activated. We will start the system by activating the roots. Thus, we execute:

MARK_ROOTS:
for all $r \in$ *roots* **do** activate r.

Lemma 2.2: *INV* holds and remains true while processes in $\mathbb{P}$ are active.

Proof: MARK_ROOTS does not introduce any black cells nor passive white cells and therefore respects *INV*. MARK1(i) eventually colors i black, but only after its children have been activated. So MARK1(i) does not cause i (nor any other node) to be a black passive node with a white passive child. □

Lemmas 1 and 2 show that *INV* is an invariant of the system.

Lemma 2.3: The system will terminate in finite time.

Proof: $\mathbb{P}$ contains 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 lemma 4.

Lemma 2.4: When the system terminates, all reachable cells are black.

Proof: First note that every node that has ever been activated (during the current round) must be black, because it has painted itself black before turning passive. It follows that all roots are black. If there is a reachable node that is still white, it follows that there must be a passive black node with a passive white child, thus contradicting *INV*. □

Lemma 2.5: When the system terminates, all black cells are reachable.

Proof: Let cell i be colored black. Then i must have been activated during the current round. Hence i is either a root (and thus reachable) or i was activated by some process MARK1(j). By repeating the argument one shows that j is reachable. As i is a child of j, i is reachable. □

Lemmas 1 through 4 prove the following theorem:

Theorem 2.1: Assume that the mutator program does not affect the current graph. When a termination detection protocol is superimposed on $\mathbb{P} = \{\text{MARK1}(i) \mid i \in M\}$, a correct graph marker is obtained.

By lemma 5 the graph marker indeed marks no garbage nodes and hence the resulting collecting algorithm collects all garbage nodes.

2.2. The basic transformation with concurrent mutator actions.

Now assume the mutator is active concurrently with the graph marking system. This implies that edges can be added or removed in an unpredictable way while the graph marker is busy. This means that the previous lemmas no longer hold. We will show how to overcome these difficulties.

First consider the deletion of edges. When edges are deleted some cells may become garbage during the marking phase, even when they were marked already. Obviously, lemma 5 no longer holds. (Its proof implicitly used the fact that no edges are removed.) This implies that there can be garbage nodes, that will not be collected. However, we can replace lemma 5 by a weaker variant:

Lemma 2.5a: When the marking phase has terminated, all black nodes were reachable at the beginning of the (current) marking phase.

Proof: Similar to lemma 5. □

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.

A more serious problem is the addition of edges, because it may lead to a violation of *INV*. The mutator may decide to add an edge between a passive black cell and a passive white one! This means that we can no longer rely on *INV* to prove the system correct. In fact the system is no longer correct, as can be shown by a straightforward counterexample. Can we modify the graph marking system so as to cope with concurrent mutator actions?

The following classical example, due to Dijkstra et. al. [10], shows that this is impossible. 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
    remove edge (a,c) ; add edge (a,c) ;
    remove edge (b,c) ; add edge (b,c)
until *false*.

Thus at any moment the data structure is in on of the three shapes shown in figure 6. Now assume the mutator always brings the graph in state 3 when the marking algorithm is visiting a, and in state 2 when the marking algorithm is visiting 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 needed to put overhead on the mutator actions. Thus we will have to require the mutator to activate nodes also, in order to maintain *INV*. When the mutator wants to add

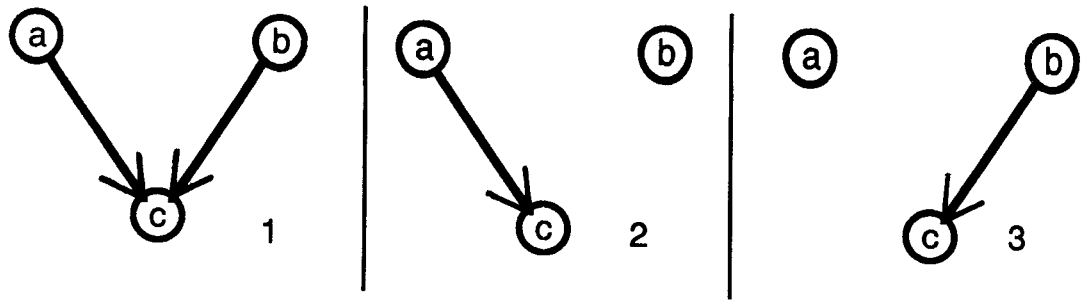

Figure 6

an edge from i to j, it executes the following routine as an "indivisible action":

ADD(i,j):
children(i) := *children(i)* + {j} ;
if i is passive black **and** j is passive white
then activate j.

Lemma 2.6: ADD(i,j) maintains *INV*.

Proof: Obvious. □

Remember now that termination detection protocols do not allow processes to become active by themselves. They must be activated by an activation message. Will termination detection protocols still work in this new situation? The following lemma helps us to answer this question in the affirmative.

Lemma 2.7: If the mutator activates a node, there is at least one other active node in the system.

Proof: The mutator can only activate a passive, white reachable node. The existence of such a node implies that the marking phase is not yet terminated and, hence, there is an active node. □

The lemma implies that a "spontaneous" activation of a cell (process) by the mutator can be modeled as the activation of the cell by the active node that must exist somewhere, and hence that a termination detection protocol must be able to handle this situation. As the proof of lemma 7 is not constructive, protocols that use acknowledgements and/or administration of

messages may give difficulties, because for these protocols it is needed that an active node is explicitly found.

Our discussion results in the following main theorem:

Theorem 2.2: When a termination detection protocol is superimposed on $\mathbb{P} = \{\text{MARK1}(i) | \ i \in M\}$ and the addition of edges is implemented as in ADD(i,j), a correct concurrent graph marker is obtained.

A few more remarks can be made.

(1) The system remains correct if the mutator always activates the passive, white targets of newly added edges, regardless of the state of their source. In fact this will only speed up the system, because some reachable nodes are activated earlier. This is done in some of the algorithms in this paper. It makes the primitive for adding edges look like

ADD(i,j):
children(i) := *children(i)* + {j} ;
if j is passive white **then** activate j.

(2) The system could also have been proven correct using the following, weaker invariant:
INV^0: Each reachable, passive node is reachable via a path in which no passive white node is the son of a passive black one.
INV^0 is maintained by the addition of edges, but not by the deletion of edges. However, it is maintained when we require the mutator to execute the following routine upon the deletion of an edge:

DELETE(i,j):
{ j is a child of i }
children(i) := *children(i)* - {j} ;
if j is passive white **then** activate j.

As in Dijkstra et. al [10] and many other papers we continue with *INV* as the basis for our collectors.

(3) Readers interested in fine-grained solutions may object, that between the two statements in ADD(i,j) or DELETE(i,j) executed by the mutator the invariant is violated, and that only later it is restored. Does this mean that this solution is not "fine-grained"? Fortunately it still is. Dijkstra et. al. [10] prove this using a weaker invariant $INV^{(1)}$, which can be informally stated as "*INV*, except in at most one place". We remark here, that it can also be proven using a weaker invariant I, defined as $INV \vee INV_0$. Informally, we can say that the mutator violates only one disjunct at a time and restores it immediately

after, so the disjunction remains valid. As mentioned in the introduction, we will concentrate only on finding algorithms and not on making them fine-grained. Thus, suppose where necessary that ADD(i,j) and DELETE(i,j) are indivisible mutator actions.

(4) MARK1(i) checks the status of i's son j before deciding whether to activate j or not. In some cases it may be more efficient to simply activate j in any case. To avoid that cells execute their code twice, we change the process code as follows:

```
MARK2(i):
    { wait until activated by external cause }
    if i was never activated before then
        begin
            for all j ∈ children(i) do
                activate j ;
            color(i) := black
        end
    stop.
```

Theorems 2.1 and 2.2 remain valid when MARK1 is replaced by MARK2. Where necessary we will write $\mathbb{P}_2 = \{\text{MARK2}(i) \mid i \in M\}$.

3. A transformational approach to the marking system of Dijkstra et. al..

One of the first on-the-fly garbage collecting systems was presented by Dijkstra, Lamport, Martin, Scholten and Steffens [10] as early as 1975. Suppose that the mutator and the collector are two processes, that operate together on one array of cells. In this section we show that the graph marker implicit in the algorithm can be derived by applying theorem 2.2, using the algorithm of Dijkstra et. al [8] as the underlying distributed termination detection protocol. We first introduce the distributed termination protocol of Dijkstra et.al. [8], prove a variant of it correct and transform it into a graph marking system.

3.1. The distributed termination detection protocol of [8].

A precise description of the algorithm by Dijkstra, Feijen and van Gasteren (hereafter called the DFG algorithm) can be found in [8]. It is assumed that processes are arranged in a logical ring. That is, each process i knows of a process $S(i)$, the *successor* of i, such that a path, starting at some node l and stepping from successor to successor, passes through every process exactly once and returns to l. Furthermore there must be a communication link from i to $S(i)$, and it is assumed that there is one unique process l that is a "leader".

The algorithm is a direct implementation of theorem 1.1. It can be seen as a distributed, sequential version of the following code:

```
repeat
    success := true ;
    for all 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. "visit p" thus means:

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

Distributing this code is easy. The leader repeatedly sends out a token on the ring, containing the value of *success*. (This is either *yes* or *no*, corresponding to *true* and *false*, respectively.) To maintain the necessary information between two visits of the token, we introduce a third state for processes, namely *blue*. The meaning of this state is "I am passive, but have been active during my current observation period". Just being passive now means that the process has been passive during the entire current observation period. So, active processes that finish their jobs initially become blue. Only after the token passes the process can become passive. Active processes do not pass the token, but wait until they have turned blue. Then they pass it on.

The leader starts sending out a *yes* token. A passive process passes the token unchanged, but a blue process passes the token as *no* to its successor. (This is the way a blue process reports the fact that it was active in the last observation period.) When the leader receives a *yes* token back and is passive itself, termination is concluded. If a *no* token is received back, a new *yes* token is sent out.

Lemma 3.1: The DFG protocol satisfies the safety condition.

Proof: Assume the leader concludes termination. No process turned the token into *no*, so all processes were passive during the entire last observation period. It is easy to see that the observation periods satisfy the conditions of theorem 1.1. □

Lemma 3.2: The DFG protocol satisfies the liveness condition.

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 passive. During the $(i+1)^{th}$ tour at the latest all processes turn to passive, 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. It is useful to consider a variant, which we call the *floating leader* variant. It is based on the following observation:

Lemma 3.3: 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 1. □

So, instead of a boolean token (*yes*/*no*) 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 now to send a token <0> on 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 in stead. It operates like this: A process that was blue when it received the token, stamps it with its identity and passes it on. Termination is concluded by a process that receives the token, stamped with its own identity.

3.2. Derivation of the graph marking system.

We will superimpose the floating leader variant of the DFG protocol onto the set of processes $\{MARK1(i)|i \in M\}$, as outlined in section 2. Each cell can be in one of 3 states as far as the DFG protocol is concerned (active, blue or passive), and in one of 2 states as far as the marking is concerned (white or black). This yields a total of 6 states, which we will represent by 4 colors according to table I.

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

Remember a process colors its associated cell black before turning passive. Hence combination (1) in table I does never occur. The statement "*color(i) := black*" in MARK1 and MARK2 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 wether 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 passive.

gray : MARK1(*i*) has been activated but did 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.

A gray node *i* can simply run the following transcription of MARK1:

for all $j \in$ *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> on 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.
- Blue processes change the token into <1>, and change themselves to black. (N.b. the value is changed to 1, because now one black process is "behind the back" of the token.)
- When the token reaches a gray node, it first waits until it turns blue, then enters it.

A process that increases the value of the token to *M* may conclude termination.

A natural choice for the successor of *i* is of course *S(i) = (i+1) mod M*, and we let the token start its journey in cell 0. The distributed termination detection algorithm is then

simulated by the following program in pseudo PASCAL (*token* denotes the value of the token, *cell* the cell it is about to visit):

```
(* Initiate the token *)
token := 0 ;
cell := M–1 ;
repeat
    cell := (cell+1) mod M ; (* Travel to next cell *)
    if color(cell) = gray then
        wait until color(cell) = blue ;
    if color(cell) = blue then
        begin token := 0 ;
            color(cell) := black
        end ;
    token := token + 1
until token = M.
```

Next we will use the same code as a scheduler for the MARK1 processes. A gray process is allowed to execute when the token is waiting to enter it, and only then. This results in the following code:

```
token := 0 ; cell := M–1 ;
repeat
    cell := (cell+1) mod M ;
    if color(cell) = gray then
        begin  (* run MARK1(cell)  *)
            for all 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 ;
    token := token + 1
until token = M.
```

Note that all gray processes will eventually be scheduled and hence the system is still guaranteed to terminate. Now we observe that a node can be blue only during the very short time between the completion of MARK1 in that node and the assignment to *color* in the

subsequent 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 can not be executed for a node if the first if-statement wasn't. Hence the statements are either executed both, or none of them is. The blue color can be eliminated when we combine the two if-statements to one. This is done in the next version of the program, where also the code for MARK_ROOTS is added:

```
(* MARK_ROOTS *)
for all r ∈ roots do color(r) := gray ;
(* initiate the token *)
token := 0 ; cell := M–1 ;
repeat
    cell := (cell+1) mod M ;
    if color(cell) = gray then
        begin
            for all j ∈ children(cell) do
                if color(j) = white then color(j) := gray ;
            token := 0 ;
            color(cell) := black
        end ;
    token := token + 1
until token = M.
```

We make a few more minor changes. The clause "**if** *color(j) = white* **then** *color(j) := gray*" can be implemented as just setting one bit. (Encode *white* as 00, *gray* as 01 and *black* as 11, and it is equivalent to "set the second bit to 1".) We call this operation "*shade(j)*". Furthermore, we can use a *decrementing* in stead of an *incrementing* counter. Of course this changes nothing in the efficiency or the correctness, but this is the method followed in [10]. The entire marking algorithm is now described by the following algorithm:

```
MARKING_PHASE:
    for all r ∈ roots do shade(r) ;
    token := M ; cell := M–1 ;
    repeat
        cell := (cell+1) mod M ;
        if color(cell) = gray then
            begin
                for all j ∈ children(cell) do shade(j) ;
                token := M–1 ;
                color(cell) := black
```

```
            end ;
            token := token − 1
        until token = 0.
```

The reader is invited to compare this algorithm to the one given in Dijkstra et. al [10]. Finally, in [10] it is not observed that it is possible to eliminate one arithmetic operation ("*token := token−1*") from the loop. In stead of using a token with a counter (the first variant of the floating leader DFG we presented) we can use a token with the id. of the last process that received the token when it was gray (cf. the remark at the end of section 3.1.). The resulting algorithm is:

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

3.3. Concurrent mutator activities.

According to theorem 2.2., the graph marker designed in the preceding subsection will also work when there is a concurrent mutator program, that executes the following code indivisibly when it adds an edge (i,j):

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

Deleting an edge can be done without any overhead. This concludes the derivation of the essential phase of the on-the-fly garbage collection algorithm of Dijkstra et. al. [10].

4. A transformational approach to the marking system by Hudak and Keller.

Hudak and Keller [12] 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 [9] on the set $\mathbb{P}_2$ of processes MARK2(i). We will discuss the Dijkstra and Scholten protocol, its transformation to the Hudak and Keller algorithm according to section 2, how concurrent mutator actions can be allowed using this algorithm, and how more concurrent mutator actions can be supported.

4.1. The distributed termination detection protocol of Dijkstra and Scholten [9].

Basically the protocol by Dijkstra and Scholten (hereafter called the DS protocol) is an acknowledgement scheme for activation messages. It is assumed that the source of all activity in the network is one special proces 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 did not yet receive 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. p is required to acknowledge all activation messages (this action is called *signaling* in [9]), 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 4.1: (Safety) When E becomes unengaged, the system is terminated.

Lemma 4.2: (Liveness) When the system is terminated, E becomes unengaged within finite time.

The algorithm is described as a very general, non-deterministic scheme (note that p is free to decide when it signals the other messages it receives). We quote from [9]:

> *We can appreciate the study of a single, highly non-deterministic algorithm as an effective way of studying a whole class of algorithms: the non-deterministic algorithm emerges when, abstracting from their mutual differences, we concentrate on what the many algorithms of this class have in common.*

It is enough for p to maintain the number of ACKs it still has to receive ($D(p)$), rather than keep a set of unACKed messages. Also, when p acknowledges a message, p need not mention the message under concern. A (further meaningless) "acknowledgement signal" suffices.

4.2. Derivation of the graph marking system.

For the purpose of deriving the graph marking system we will 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 will mean: send an activation message, containing the identity *father* of the sender to i. We will denote this message as <ACT, i, *father*>. Its receipt by i triggers execution of the following pseudo PASCAL code, which contains the code for MARK2(i) as well as the code for the DS protocol:

```
ACTIVATE(j, father):
    if j was not activated before then
        begin
            C(i) := 1 ;
            (* now run MARK2 *)
            for all 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)* of course means: send an acknowledgement to the node *father*.

The complete graph marking algorithm that we can now derive is "message driven", i.e., something can happen only upon the receipt of a certain message. We can 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. State information about the process is stored explicitly. So, suppose node i contains yet another extra field *father(i)*. We present the message driven form:

```
ACTIVATE(i, father): (* executed upon receipt of <ACT, i, father> *)
    if i was not activated before then
        begin
            C(i) := 1 ;
            father(i) := father ;
            for all 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 upon receipt of a signal by i *)
    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. It is easy to see that this minor change does not affect the correctness or termination properties of the algorithm. We do this, so the statements "*color(i) := black*" and "*C(i) := 0*" appear always together. Soon we will replace these two statements by one, similar to what we did in section 3.2. Each node can be in one of three states only (except between the two statements mentioned). These states are:

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 3 we will 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
            color(i) := gray ;
            father(i) := father ;
            for all 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 *)
    for all r ∈ roots do
        begin activate(r,E) ;
            D(E) := D(E)+1
        end ;
    while D(E)>0 do
        begin receive a signal ;
```

D(E) := D(E)–1

end.

Of course this part of the algorithm can be written in message driven form also. This part is somewhat underdeveloped in [12]. The original algorithm was suited only for graphs with one root.

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 [12] suitable "locking machinery" is built in to guarantee exclusive access. This machinery is ignored here.

4.3. 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 [10] handles it. The invariant *INV*, defined in section 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 black, no child of i is white.

The mutator must not violate this invariant and thus it must activate nodes sometimes. However, in order to enable a node to send an acknowledgement later, it must be given a father. So the mutator must be able to find a gray node that is willing to "adopt" the node. And, alltough lemma 2.7 guarantees that such a node exists, it is not trivial to find one. This is why in [12] the behaviour of the mutator is limited where the addition of edges is concerned. Only in a small number of specific cases edges may be added.

One of these cases and its solution (cf. [12]) is the following: it is allowed to add an edge to a grandson by means of a suitable construction *add_grandson(a,b,c)*, where it is assumed that edges (a,b) and (b,c) do exist already. See fig. 7. Obviously the operation *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 this table:

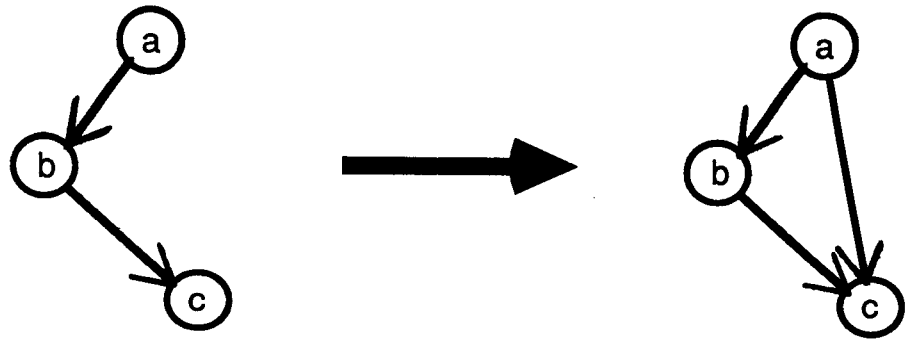

Figure 7

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

We consider these cases in turn:

Case 1: Suppose *a* is gray and *b* is white. Then an activation message has been sent to *b*, but clearly *b* did not yet send one to *c*. Hence it is not certain that an activation message has been sent to *c* at all. In order to maintain invariant A, we will have to send *c* an activation message and here we can make *a* into *c*'s father, i.e., *a* adopts *c*.

Case 2: Suppose *a* is black. Then *a* is not allowed to have a white child by B. But, it is also not allowed to send activation messages because it is not engaged. So in this case we force *b* to adopt *c*. Before we can make the link, we must wait until the activation has resulted in *c* becoming gray: according to B, it is not enough that an activation message is on its way to *c*. (The reader is invited to construct an example to show the unsafety of the system if the link is made too fast.) The operation *add_grandson* can thus safely be implemented by means of the following program in pseudo PASCAL:

ADD_GRANDSON(*a*,*b*,*c*)

(* *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}.
```

Several other mutator primitives can be supported in a similar way, see [12] or Hudak's thesis [13].

4.4. 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 wants to add an arbitrary edge (a,c). Overhead is not always needed. 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, the preceding action is impossible and yet c must be linked and 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. Next we have to wait until this message is processed by c and c is (at least) gray.

Concluding, 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 only 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 4.1). 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 bottle neck. 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. 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, they can ask their neighbors etc.

(3) We can add some "special purpose" nodes to the data structure. Suppose there is one special root S_p on each processor p. S_p will never have "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 colecting process.

   This sounds as if we should superimpose yet another termination detection protocol on the system, which seems undesirable.

5. A highly parallel garbage collector.

Recall the collection system from section 3. As in section 3 we will assume here that the mutator is one sequential process, running on a non-distributed memory. In this section we will introduce a highly parallel garbage collector for it, i.e. a collector that consists of any number of garbage collecting processes. The collector and its underlying termination detection protocol are generalizations of those in section 3.

5.1. A highly parallel termination detection protocol.

The DFG protocol basically consists of sequentially visiting all processes. If all processes had been passive all the time since the last visit, termination can be concluded by theorem 1.1. 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 ;
        for all processes p do
            begin
                wait until p is not active ;
                if p is blue then
                    begin success := false ;
                        p becomes passive
                    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. Tan and van Leeuwen [20] exploit this observation and derive some very nice, general termination detectors.

The basis of the on-the-fly garbage collection algorithms in this section will be the following termination detection protocol skeleton:

```
repeat
    success := true ;
    for all i do "visit i"
until success.
```

Seen in this light the step to the floating leader version of the DFG protocol is rather intricate. Only when one uses the fact that the processes are visited sequentially and in the same order each time, it makes sense to combine the "tail" of one tour with the "head" of the next tour to obtain a "succesful round". We hope the reader will find this presentation of the floating leader variant less convincing then the one given in section 3.1., because this proves that the way in which an algorithm is presented can either help or block the human mind in finding changes and improvements on it. The "floating leader" variant of the "T–protocol" of Tan and van Leeuwen [20] is of a totally different nature: it fits in the above skeleton, where the floating leader DFG protocol does not.

5.2. 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 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 3, the code for MARK_ROOTS is of course:

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

A "visit" will have about the same semantics as in section 3. 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, more explicitly:

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

We will supply two parallel visiting schemes to complete the garbage collecting systems. First, 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 in k parts, and give each garbage collection processor a part. So, let $\{S_i | 1 \leq i \leq k\}$ be a partition of M. The marking system is given by:

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

This visiting scheme has the disadvantage that the partition of the memory cells must be fixed in advance. Thus it is not possible here to adapt the work load of a processor to its speed dynamically. Such an adaption is possible and can be elegantly implemented for a system consisting of only two garbage collecting processors, as is done in the following visiting scheme. Let the two processors start at one end of the cell array each, and work towards each other. When they meet somewhere, the round is completed. Call the two garbage collecting processes GCA and GCB. We can describe the two processes as follows:

GCA:

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

GCB:

```
repeat
    synchronize ;

    b := M-1 ;
    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 overhead, and (2) make a dynamic scheme for more than two processors.

5.3. Concurrent mutator actions.

According to theorem 2.2, the graph marking algorithms will also work when there is a concurrently active mutator process that executes the following code when it adds an edge:

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

No overhead for deletion of edges is needed.

6. On-the-fly garbage collection in a ring of processors.

Suppose the system consists of a number of processors, each with its own local memory. On each of the processors one or more mutator processes and a garbage collecting process can be active (i.e., the model is that of section 4). If it is possible to arrange the processors in a logical ring, the same can be done with the memory cells, see fig. 8. We will show that the paradigm of section 2 can be used to derive efficient on-the-fly garbage collectors for such systems. As the underlying distributed termination detection protocol we use the floating leader DFG protocol, as introduced in section 3.1.

6.1. The graph marking system for a ring of processors.

Like in section 3.2., the circulation of the token around the ring of nodes has a scheduling function as well as a termination detection purpose. But, we want all garbage collecting processes to do work, and not only the one that keeps the token. So we separate part of the scheduling of the MARK1 processes from the termination detection process. Each processor that finds a gray node in its memory can run MARK1 for this node. Of course it is then necessary to reintroduce the color blue, to represent nodes that have been active since the last visit of the token. Assume each processor knows about the total number M of memory cells, that processor p has M_p cells locally addressable as $1..M_p$ and that *shade(j)* is done by sending a "shading message". We assume that shading messages reach their destination in time 0. A token traveling along the array of cells in a processor is simulated by a pseudo PASCAL program (see module 4 below) much as we did in earlier sections. When it crosses an "interprocessor border" in the node ring the token is actually passed as an interprocessor message. The complete marking system consists of 4 "modules" (*P* denotes any garbage collecting

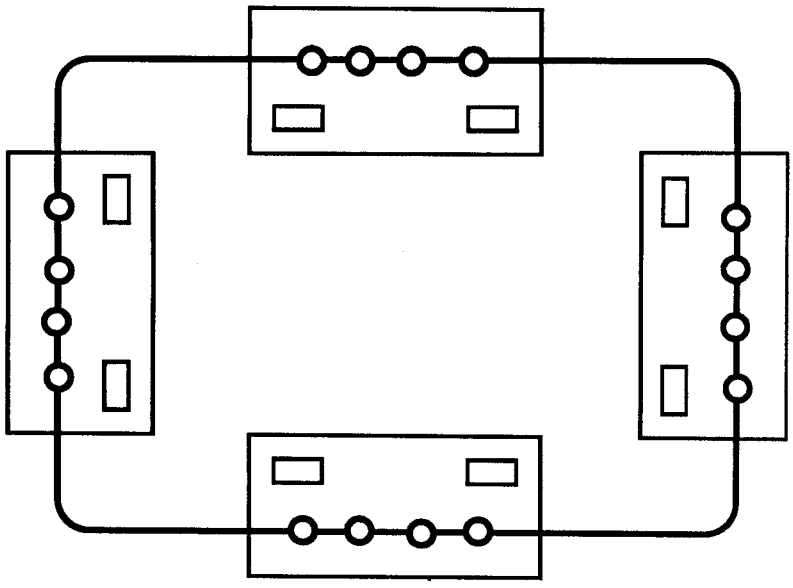

Figure 8

processor):

1. (* MARK_ROOTS *)
   all *P* **do**
   for all *r* ∈ *roots* **do** *shade(r).*

2. (* Local progress of marking *)
   all *P* **do**
   repeat
   i := ... ; (* a gray node in processor *P* *)
   for all *j* ∈ *children(i)* **do** *shade(j)* ;
   color(i) := *blue*
   until there is termination.

3. (* Start of DFG protocol *)
   (* Only one processor, the leader *l*, executes: *)
   send <0> to *S(l).*

```
4. (* Scheduling and termination detection *)
   (* Upon receiving the token <v>, P executes: *)
      val := v ;
      for i:=1 to M_P do
            begin
                  if color(i) = gray then
                        begin
                              for all j ∈ children(i) do shade(j) ;
                              color(i) := blue
                        end ;
                  if color(i) = blue then
                        begin
                              val := 0 ;
                              color(i) := black
                        end ;
                  val := val+1 ;
                  if val=M then (* there is termination *)
                        begin (* take appropriate action *) end
            end ;
      send <val> to S(P).
```

6.2. Concurrent mutator actions.

Again this marking system allows concurrent mutator activities, as long as the mutator executes the code for ADD(i,j), as defined before, when it adds an edge. Here this code must be indivisible, it is not safe to allow interleaving in a finer grain. Recall remark 3 at the end of section 2.2. When there are 2 (or more) mutator processes active in parallel, each of them can violate one disjunct, so I becomes violated. Also the correctness proof in [10] is not adaptable to the case of $k \geq 2$ parallel mutators. It is easy to find an example to show the unsafety of the system if the ADD operation is not indivisible. The idea to reverse the order of the two operations in ADD intuitively ensures that "the invariant is restored before it is violated" and hence "is not violated at all". This however leads to an unsafe algorithm even in the case of only one sequential mutator process. The system becomes vulnerable to the "Woodger-bug", see [10]. Putting overhead on the deletion of edges in stead of the insertion will not resolve these problems.

We therefore suggest to use a locking mechanism that ensures that the ADD operation is indivisible, at least from the collector's point of view. The construction of a garbage collector

that allows multiple mutator processes to run concurrently with it and has a "grain of action" as big as single bit operations (like the collectors of Ben-Ari [2] and Dijkstra et.al. [10] have) remains an interesting subject of further research.

7. On-the-fly garbage collection in an arbitrary network.

In this section we assume the same architecture as in section 6, except that the processes are arranged in an arbitrary, connected network rather than in a ring. We describe a hierarchical graph marker and extend the approach to the on-the-fly garbage collector due to Hughes [14].

7.1. Hierarchical termination detection.

The distributed termination detection problem is usually considered in a *totally distributed* environment. There is no global control and processes can communicate by message transfer only. When, like in e.g. section 6 and some of our other algorithms, a huge number of processes reside in one physical processor, it is possible to make an extra process in every processor that acts as an "operating system" or "supervisor" for the (basic) processes residing in that processor. This supervisor has a sort of global control over these processes: it can see at any time whether one of them is active, because everything that happens here is "local" to the operating system.

This suggests the following notions. Let $\mathbb{P}$ be a set of processes as in section 1.3., and let P, Q, R,.. be *groups* (=sets) of processes, such that $\{P, Q, R,...\}$ is a partition of $\mathbb{P}$. We say that a *group is active* if at least one of its members is active, and *passive* if none of its members is active. Note, that activity of a group is decided by observing group members only.

Lemma 7.1: A passive group P can become active only if one of its members receives an activation message from a member of another group Q that is still active.

Proof: Suppose a passive group P becomes active. This means that one of its members, say x, was activated. So x received an activation message from some active process y. All processes in P were passive, so y must be a member of some other group Q. (Note: we used that activation messages, at least within one group, have transmission delay 0.) □

When the situation of theorem 7.1. arises, we say that P is activated by Q. Representing a group P by a process S_P, we can formulate a two-level hierarchical termination detection protocol as follows:

(1) A low-level protocol ensures that
S_P is active $\Leftrightarrow$ some $x \in S_P$ is active,

(2) A high-level protocol detects when $\{S_P, S_Q, ...\}$ is terminated.

7.2. A hierarchical graph marker for general networks.

The ideas presented in 7.1 can be transformed into a graph marker in many ways. Suppose each garbage collecting processor (i.e., each S_P) maintains a queue of currently active cells. Passivity of the group is equivalent to the queue being empty. Receipt of a "*shade(j)*" message triggers:

if *j* is white **and** *j* not in queue yet
then enqueue *j*.

(and eventually an acknowledge for the message). Further, S_P executes active MARK1 processes by the following code:

```
while queue not empty do
    begin
        dequeue(i) ;
        for all j ∈ children(i) do shade(j) ;
        color(i) := black
    end.
```

The processors can run any arbitrary termination detection protocol to detect the termination of the marking phase. For termination detection protocols on arbitrary networks, see e.g. Tan and van Leeuwen [20] or Tel [21].

7.3. Hughes' garbage collector.

In [14] Hughes presents a garbage collector for a multiprocessor environment. We now show that some of its underlying ideas can be derived by applying the techniques presented in this paper.

Hughes runs infinitely many marking algorithms at the same time, namely one copy of the marking algorithm for each time t. Call this copy F_t. The color field in the nodes is

replaced by a *timestamp*. A node having timestamp t_n can be thought of as having the following color:

- black for all F_t, $t \leq t_n$,
- white for all F_t, $t > t_n$.

A node, created at time t, gets t as its timestamp and is thus marked for all current marking phases. Activation messages are also timestamped. An activation message with stamp t_a is denoted by <ACT, t_a> and means "this is an activation message for all F_t, $t \leq t_a$". Its receipt triggers the following transcription of MARK2:

```
{ i receives <ACT, t_a> }
if t_a > timestamp(i) then
    begin
        for all j ∈ children(i) do
            send <ACT, t_a> to j ;
        timestamp(i) := t_a
    end.
```

In section 7.2 we saw that each processor had to keep track of its active nodes and knew at any time, *whether* it was active or not. Here each processor P has to keep track of *the smallest* t for which it has more work to do for F_t. Hughes calls this value $redo_P$. He runs an (infinite) number of classical termination detection protocols to determine which F_t can be considered as terminated. The protocol he uses is Rana's [17]. In this way he has constructed one of the first algorithms for approximation of Global Virtual Time (GVT). Rana's protocol has some disadvantages, namely that processors must be arranged in a ring (as in section 6) and that messages must travel instantaneously. For more GVT algorithms on general graphs and under more general assumptions about communication, see Tel [21].

If P knows *minredo*, the global minimum of all values *redo* of all processors, P can collect all cells with a timestamp smaller than *minredo*. Because many phases overlap, the recycling of cells has become a *continuous* process rather than a periodical one, like in the previous garbage collectors, in which a large number of cells is recycled every now and then. Thus, the system acts an an *incremental garbage collector*.

8. Discussion and conclusions.

In this paper we have exploited an observation, originally due to Scholten, that there is an intimate connection between on-the-fly garbage collection and termination detection. We demonstrated how concurrent graph marking algorithms, the main ingredient of on-the-fly garbage collectors, can be obtained by a rather mechanical transformation from distributed termination detection protocols. Using this transformation we have obtained clear and transparant derivations of several old and new on-the-fly garbage collecting algorithms, and could sometimes even improve on some of the old algorithms. We have not investigated "granularity" aspects deeply. The case studies show that our approach is useful, and apparently of sufficient generality to deal with the design of all known on-the-fly garbage collectors.

Our aim has been to contribute to a deeper understanding of what "distributed computation" means. We think program transformation techniques help us to realize that the number of essentially different distributed algorithms is not as large as it seems. It might turn out that distributed algorithms, and the problems they solve, are composed of a relatively small number of basic "modules".

In this paper we have made the connection between distributed termination detection and on-the-fly garbage collection concrete. Other researchers have established similar connections between termination detection and deadlock detection (see Natarajan [16]), election (see Tan and van Leeuwen [20]) or "Global Virtual Time" (see Tel [21]). Studying the problem of distributed termination detection has lead to several new models of distributed computations (see Shavit and Francez [19] or Beilken et. al. [2]). Hence it seems worthwile to make a more thorough study of the mechanisms underlying distributed termination detection.

Acknowledgement: We thank C.S. Scholten for a number of useful comments on an early version of this paper.

9. References.

[1] K. R. Apt, Correctness proofs of distributed termination algorithms, Rep. 84-51, LITP, Université paris 7, Paris, 1984.

[2] C. Beilken, F. Mattern, and M. Reinfrank, Verteilte Terminierung - ein wesentlicher Aspekt der Kontrolle in verteilten Systemen, SFB124 Bericht Nr 41/85, Universität Kaiserslauten, dec 1985.

[3] M. Ben-Ari, Algorithms for on-the-fly garbage collection, ACM ToPLaS 6 (june 1984), pp. 333-344.

[4] K.M. Chandy, and J. Misra, A paradigm for detecting quiescence properties in distributed computations, Report TR–85–02, Dept of Comp. Sc., University of Texas at Austin, Austin, Texas, 78712.

[5] S. Cohen, and D. Lehmann, Dynamic systems and their distributed termination, in: Proc. 1st Annual ACM Symp. on Principles of Distributed Computing, Ottawa, 1982, pp.29-33.

[6] J. Cohen, Garbage collection of linked datastructures, Comp. Surv. 13 (sept. 1981) pp. 341-367.

[7] E.W. Dijkstra, On making solutions more and more fine-grained, EWD622, in: E.W. Dijkstra, Selected writings on computing, a personal perspective, Springer Verlag, Heidelberg 1982.

[8] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren, Derivation of a termination detection algorithm for distributed computations, Inf. Proc. Lett. 16 (june 1983), pp. 217-219.

[9] E.W. Dijkstra, and C.S. Scholten, Termination detection for diffusing computations, Inf. Proc.Lett. 11 (aug. 1980), pp. 1-4.

[10] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, On-the-fly garbage collection : An exercise in cooperation, Comm. ACM 21 (nov 1978), pp. 966-975.

[11] N. Francez, Distributed termination, ACM ToPLaS 2 (1980), pp. 42-55.

[12] P. Hudak, and R.M. Keller, Garbage collection and task detection in distributive applicative processing systems, in : Proceedings of the ACM Symp. on LISP and functional programming, Pittsburg, 1982.

[13] P. Hudak, Distributed graph marking, Research Report # 268, Dept. of Computer Science, Yale University, New Haven, 1983.

[14] J. Hughes, A distributed garbage collection algorithm, in : J.P. Jouannaud (ed.), Functional Programming Languages and Computer Architecture, LNCS vol. 201, Springer Verlag, Heidelberg, 1985, pp. 256-272.

[15] H.T. Kung, and S.W. Song, An efficient parallel garbage collection system and its correctness proof, in: Proceedings 18th Annual IEEE Symp. on Found. of Comp. Sc., 1977, pp. 120-131.

[16] N. Natarajan, A distributed scheme for detecting communication deadlock, IEEE Trans. on Softw. Eng. SE 12 (apr. 1986), pp. 531-537.

[17] S.P. Rana, A distributed solution to the distributed termination problem, Inf. Proc. Lett. 17 (1983) pp. 43-46.

[18] H. Schorr, and W.M. Waite, An efficient machine-independant procedure for garbage collection in various list structures, CACM 10 (1967) pp501-506.

[19] N. Shavit, and N. Francez, A new approach to detection of locally indicative stability, in L. Kott (ed), Proceedings 13th ICALP, Springer Verlag LNCS 226.

[20] R.B. Tan, and J. van Leeuwen, General symmetric distributed termination detection, Techn. Rep. RUU-CS-86-2, Dept. of Computer Science, University of Utrecht, 1986. (Submitted to Computers and Artificial Intelligence).

[21] G. Tel, Distributed Infimum Approximation, Techn. Report RUU–CS–86–12, Dept. of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1986.

[22] B. Wegbreit, A space efficient list structure tracing algorithm, IEEE Trans. Comput., vol C21, sept. 1972, pp1009–1010.