THE DERIVATION OF GRAPH MARKING ALGORITHMS FROM DISTRIBUTED TERMINATION DETECTION PROTOCOLS*

Gerard TEL**, Richard B. TAN*** and Jan van LEEUWEN

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

Communicated by L. Lamport
Received October 1986
Revised June 1987

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. and of Hučak and Keller. The approach also leads to several new, highly parallel algorithms for concurrent garbage collection. We also analyze a garbage collecting system due to Hughes from our current perspective.

1. Introduction

In the past 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 private 'ad-hoc' idea and that there is hardly any common background in these algorithms.

In this paper we aim at a general methodology for deriving on-the-fly garbage collection algorithms. We will concentrate on the structure of the algorithms, and not on aspects of granularity. 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.

* 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).
** UUCP address: \mcvax\ruinf\vaxgerard. Work supported by the Netherlands Organization for the Advancement of Pure Research (ZWO).
*** Department of Mathematics and Computer Science, University of Sciences and Arts of Oklahoma, Chickasha, OK 73018, U.S.A.

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 on-the-fly garbage collection 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. We then give applications of this transformation in the next few sections. In Section 3 we apply the transformation to the termination detection protocol of Dijkstra et al. [8]. We 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 generalize this approach and derive some highly parallel graph marking algorithms. In Section 5 we apply the transformation to the termination detection protocol of Dijkstra and Scholten [9] to obtain the graph marking algorithm of Hudak and Keller [12]. In Section 6 we present a new parallel graph marking algorithm, based on the distributed termination detection protocol by Dijkstra et al. [8]. In Section 7 we outline a two-level 'hierarchical' termination detection protocol and develop the corresponding graph marking algorithm. The on-the-fly garbage collection algorithm presented by Hughes [14] is an intricate refinement of this algorithm. In Section 8 we conclude with 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 or deletes edges to or from garbage nodes. New nodes are allocated 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. 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 will use the words cell and node interchangeably. A cell $i$ can have several fields (representing the data), among which is a field $\text{children}(i)$, 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:

\[
\text{ADD}(i, j):
\]
\[
(* \ i \ \text{and} \ j \ \text{are reachable nodes} *)
\]
\[
\text{children}(i) := \text{children}(i) + \{j\}
\]

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

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 $\text{color}$ 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 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 is 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 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. [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. 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, processor dedicated to garbage collection only. In this computational model there are two processors working on the same data in an ‘independent’ manner. The actions of the two processes are 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 (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. 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’).

1.2. Distributed termination detection

Distributed termination detection is the problem of determining when all activity in a distributed system of processes has ceased. Assume each member of a set of processes is performing a certain task, which it will eventually finish. During its work, a process may decide to send new tasks to other processes. It is possible that in this way a process is ‘awakened’ again, after having been passive for a while. It is usually assumed that no new task is created in a passive process after the initialization of the system and that only active processes can wake up passive ones. Clearly, when all processes 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 passive.

A more formal definition of the distributed termination detection problem is as follows. Let $P$ be a set of processes, each of which can be in one of two states, namely active or passive. Only active processes may send so-called activation messages to other processes. A process may change state, with the restriction that a change from passive to active may take place only upon receipt of an activation message. We impose no restrictions on state changes from active to passive. We say the system $P$ is terminated when all processes in $P$ are in the passive state and there are no activation messages under way. We say the terminated state is stable.

**Theorem 1.1.** When the system is terminated, it remains terminated.
Proof. Assume the system is terminated. No process will become active, because it can do so only upon receipt of a message, and there is no message. No message will be sent, because this can be done only by active processes, and there are no active processes. So the system will remain terminated.

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 $P$ to enable them to do so. A 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 the time being that the sending and receiving of a message are instantaneous. We illustrate the behavior of the processes in $P$ by time diagrams (see e.g. Fig. 1), in which each horizontal line represents the behavior of one process in $P$. The horizontal axis represents time. By a fat line (———) we indicate that a process is active and by an arrow (↓, ↑) we mean the exchange of an activation message.

If every process reports its state instantaneously at time $t$ to some central site, these reports would enable one to determine the state of the system at time $t$. For example, in Fig. 1, at time $t_1$ some processes report an active state and hence the system is not terminated. At time $t_2$ 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. 2): process $p_1$'s clock is fast and $p_1$ reports its passive state somewhat before time $t$. Then time $p_2$ 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]:

**Theorem 1.2.** Observe each process \( p_i \) during an observation interval \((start_i, end_i)\) and ensure that for all \( i, j \): \( start_i < end_i \) 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 time \( t \) and there were no messages under way. Hence the system was terminated at time \( t \), and thus remains so thereafter (Theorem 1.1). \( \square \)

Theorem 1.2 enables one to give correctness proofs of most of the termination detection protocols we use in this paper, like the protocols due to Dijkstra et al. [8], Tan and Van Leeuwen [20], and the variants we use of these protocols.

It is possible to adapt the protocols (and Theorem 1.2) to the more realistic situation that activation messages actually take some time to reach their destination. This is not done here, but it is done, 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 \( color(i) = \text{white} \) 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, which means that the data structure is fixed. We will subsequently adapt the mutator program so as to run concurrently with the marker safely. Also, we assume that communication between processes is immediate.
2.1. The basic transformation

For each cell \( i \) in the computer's memory we introduce the (conceptual) process \( \text{MARK1}(i) \), defined as

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

Let \( M = \{0, 1, \ldots, N - 1\} \) be the set of cells in the computer's memory and define the set of processes \( \mathcal{P} \) by \( \mathcal{P} = \{\text{MARK1}(i) | i \in M\} \). 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. For an edge \( e \) from \( p \) to \( q \), we say \( p \) is the source and \( q \) is the target of \( e \).

**Definition 2.1.** \( \text{INV} \) is the property that for all edges, the source is not passive black or the target is not passive white.

**Lemma 2.2.** \( \text{INV} \) holds in the initial state of the system.

**Proof.** Obvious, because there are no black cells in the initial state. \( \square \)

Of course, nothing happens unless processes are activated. The system is started by execution of the following code:

\[
\text{MARK_ROOTS:} \\
\text{for all } r \in \text{roots do activate } r.
\]

Some scheduling mechanism will be assumed or provided for the execution of the set of processes.

**Lemma 2.3.** \( \text{INV} \) remains true (while processes in \( \mathcal{P} \) execute).

**Proof.** \( \text{INV} \) is in danger only if

1. an edge is added from a passive black to a passive white node,
2. the source of an edge becomes passive black, or
3. the target of an edge becomes passive white.

However, in all cases \( \text{INV} \) remains true.

1. We assumed no concurrent mutator actions, hence this does not occur (but, see Section 2.2);
2. Node \( i \) becomes passive black after execution of \( \text{MARK1}(i) \). If \( i \) is the source of some edge \( (i,j) \), \( j \) was activated by \( \text{MARK1}(i) \), so the target of the edge is not passive white;
3. A node that is not passive white never becomes passive white thereafter. \( \square \)
Lemmas 2.2 and 2.3 show that INV is an invariant of the system.

**Lemma 2.4.** Under the assumption that an active process will eventually be scheduled and execute its next statement, the system will terminate.

**Proof.** $P$ consists of only a finite number of processes. Each of them is activated at most once and runs to completion in finite time. \(\square\)

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

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

**Proof.** When the system has terminated all nodes will be either passive white or passive black. Every node that has ever been activated (during the current round) is black, because it has painted itself black before turning passive. It follows that all roots are black. A child of a black node is not active (by termination of $P$) and not passive white by INV. It follows that children of black cells are black. By induction it follows that all reachable cells are black. \(\square\)

**Lemma 2.6.** When the system terminates, all black cells are reachable.

**Proof.** Because no deletions of edges occur, reachable cells remain reachable. To prove that no garbage cells are activated, assume $j$ is the first garbage cell that is activated. $j$ is not a root (because roots are reachable) and hence $j$ was activated by the execution of $\text{MARK1}(i)$ for some $i$. So, cell $i$ was activated earlier than $j$. Thus, by the choice of $j$, $i$ is reachable and, by the definition of $\text{MARK1}(i)$, $j$ is a child of $i$. Hence $j$ is reachable, a contradiction. A black cell must have been activated, hence it is reachable. \(\square\)

Lemmas 2.2–2.5 prove the following theorem:

**Theorem 2.7.** When a termination detection protocol and a scheduling mechanism are superimposed on $P = \{\text{MARK1}(i) | i \in M\}$, a correct graph marking algorithm is obtained.

By Lemma 2.6 the graph marker indeed marks no garbage nodes and hence the resulting collecting algorithm collects all garbage nodes in one round.

### 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. We will see how this affects the results of Section 2.1.
We first consider the deletion of edges. When edges are deleted, cells may become garbage during the marking phase even when they were marked already, and Lemma 2.6 no longer holds. (Its proof used the fact that no edges are removed.) So there can be garbage nodes that will not be collected. However, we can replace Lemma 2.6 by a weaker variant:

**Lemma 2.6a.** When the marking phase has terminated, all black nodes were reachable at the beginning of the (current) marking phase.

**Proof.** A cell that is black was activated during the execution of the marking phase. Suppose \( j \) is the first cell that is activated but was not reachable at the beginning of the marking phase. As in the proof of Lemma 2.6 it follows that \( j \) is not a root and \( j \) was activated by some \( \text{MARK}\_1(i) \). By the choice of \( j \), \( i \) was reachable at the beginning of the current marking phase. If the edge \((i, j)\) existed at that time, \( j \) was reachable at that time also. If the edge \((i, j)\) was added later, \( j \) was reachable at the time of the addition of the edge, hence it was reachable at the beginning of the marking phase also. \( \square \)

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 \DELET\E primitive given in Section 1.1 does not violate the graph marker's invariant \( \text{INV} \). 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 \( \text{INV} \). In this case the system as introduced in the previous subsection is no longer correct. The following classical example, due to Dijkstra et al. [10], 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

\[
\text{repeat}
\begin{align*}
\text{DELETE} & (a, c); \text{ADD} (a, c); \\
\text{DELETE} & (b, c); \text{ADD} (b, c)
\end{align*}
\text{until false.}
\]

Thus at any moment the data structure is in one of the three states shown in Fig. 3. Now assume the mutator always brings the graph into state 3 when the marking algorithm is inspecting \( a \), and in 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 overhead on the mutator actions. We will require the mutator to activate nodes also, in order
to maintain \( INV \). From now on addition of edges is done by (indivisibly) executing the following code:

\[
\text{ADD}(i, j):
\]

\[
(* \ i \ \text{and} \ j \ \text{are reachable} *)
\]

\[
\text{children}(i) := \text{children}(i) + \{j\};
\]

\[
\text{if} \ i \ \text{is not passive white} \ \textbf{and} \ j \ \text{is passive white} \ \textbf{then} \ \text{activate} \ j.
\]

**Lemma 2.8.** \( \text{ADD}(i, j) \) now maintains \( INV \).

**Proof.** The added edge does not lead from a passive-black to a passive-white node because of the (eventual) activation of \( j \). \( \square \)

In the proof of stability of the termination condition (Theorem 1.1) it was used that only active processes send activation messages. We must prove that this stability still holds under activations by the mutator.

**Lemma 2.9.** If the mutator activates a node, the marking system is not terminated.

**Proof.** The mutator can activate a passive, white reachable node. The existence of such a node implies that the marking phase is not yet terminated, by Lemma 2.5. \( \square \)

It follows that the termination condition as defined in Section 1.1 is stable. But, it is not certain that existing termination detection protocols will always work correctly under spontaneous activation by the mutator. Lemma 2.9 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 2.9 is not constructive, termination detection protocols that use acknowledgments and/or administration of messages may give difficulties. In these protocols it is necessary that an active node is explicitly found (see Section 5).

The discussion results in the following main theorem:

**Theorem 2.10.** When a termination detection protocol and a scheduling mechanism are superimposed on \( \mathcal{P} = \{\text{MARK1}(i) \mid i \in M\} \), and the addition of edges is implemented as in \( \text{ADD}(i, j) \), a correct concurrent graph marker is obtained.
Some further 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

\[
\text{ADD}(i, j):
\]

\[
\text{children}(i) := \text{children}(i) + \{j\};
\]

\[\text{if } j \text{ is passive white then activate } j.\]

(2) 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 regardless of this. To avoid that cells execute their code twice, we change the process code as follows:

\[
\text{MARK2}(i):
\]

\[\{\text{wait until activated by external cause}\}\]

\[\text{if } i \text{ was never activated before then}\]

\[\text{begin}\]

\[\text{for all } j \in \text{children}(i) \text{ do}\]

\[\text{activate } j;\]

\[\text{color}(i) := \text{black}\]

\[\text{end}\]

\[\text{stop.}\]

Theorems 2.7 and 2.10 remain valid when MARK1 is replaced by MARK2. Where necessary we will write \(P_2 = \{\text{MARK2}(i) | i \in M\}\).

(3) In this section we assumed immediate arrival of activation messages. This is realistic if all cells reside in the memory of one computer. In some models this may not be realistic. In such cases immediate communication can be simulated logically, or the theory in this section can be modified for asynchronous communication. The invariant

\[\text{INV': For all edges, the source is not passive black, or the target is not passive white, or an activation message is under way to the target}\]

can be used. This invariant is used in Section 5. Note that under synchronous communication it is equivalent to the earlier invariant.

3. A transformational approach to the algorithm of Dijkstra et al.

One of the first on-the-fly garbage collecting algorithms was presented by Dijkstra et al. [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
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 1. Remember that a process colors its associated cell black before turning passive. Hence combination (1) in Table 1 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 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.

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

```plaintext
for all j ∈ 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;
- Gray processes keep the token until they become blue, and then act as a blue process;
- 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 will 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 and efficient graph marking

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>marking colors</td>
</tr>
<tr>
<td>detector states</td>
</tr>
<tr>
<td>active</td>
</tr>
<tr>
<td>blue</td>
</tr>
<tr>
<td>idle</td>
</tr>
</tbody>
</table>
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):

\[
\begin{align*}
(* \text{ Initiate token } *) & \\
\text{token} & := 0; \text{cell} := N - 1; \\
(* \text{ Circulate token } *) & \\
\text{repeat} & \\
(* \text{ Travel to next cell } *) & \\
\text{token} & := \text{token} + 1; \text{cell} := (\text{cell} + 1) \mod N; \\
(* \text{ Wait if cell is gray } *) & \\
\text{if} & \ \text{color(cell)} = \text{gray} \text{ then} \\
& \text{wait until} \ \text{color(cell)} = \text{blue}; \\
(* \text{ Token becomes 0 if cell is blue } *) & \\
\text{if} & \ \text{color(cell)} = \text{blue} \text{ then} \\
& \text{begin} \ \text{token} := 0; \ \text{color(cell)} := \text{black} \ \text{end} \\
\text{until} & \ \text{token} = N.
\end{align*}
\]

In the second step we add the scheduling and execution of MARK1 processes. We must ensure that every gray MARK1 process will eventually execute 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. This results in the following code:

\[
\begin{align*}
\text{token} & := 0; \ \text{cell} := N - 1; \\
\text{repeat} & \\
\text{token} & := \text{token} + 1; \ \text{cell} := (\text{cell} + 1) \mod N; \\
\text{if} & \ \text{color(cell)} = \text{gray} \text{ then} \\
& \text{begin} \ (* \text{ execute MARK1(cell) } *) \\
& \ \text{forall} \ j \in \text{children(cell)} \ \text{do} \\
& \ \ \text{if} \ \text{color(j)} = \text{white} \ \text{then} \ \text{color(j)} := \text{gray}; \\
& \ \ \text{color(cell)} := \text{blue} \\
& \text{end;} \\
\text{if} & \ \text{color(cell)} = \text{blue} \text{ then} \\
& \text{begin} \ \text{token} := 0; \ \text{color(cell)} := \text{black} \ \text{end} \\
\text{until} & \ \text{token} = N.
\end{align*}
\]

Note that all gray processes will eventually be scheduled and hence the system is still guaranteed to terminate. In the next 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 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 the statements are either both executed, or none 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 the code for MARK_ROOTS is also added and the 'shorthand' shade(j) is introduced for "if color(j) = white then color(j) := gray":

\[
(* \text{MARK\_ROOTS *})
\]
for all \( r \in \text{roots} \) do shade(\( r \));
\( \text{token} := 0 \); \( \text{cell} := N - 1 \);
repeat
\( \text{token} := \text{token} + 1 \); \( \text{cell} := (\text{cell} + 1) \mod N \);
if \( \text{color(cell)} = \text{gray} \) then
begin forall \( j \in \text{children(cell)} \) do shade(\( j \));
\( \text{token} := 0 \); \( \text{color(cell)} := \text{black} \)
end
until \( \text{token} = N \).

(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. [10] and note the similarities. In [10] 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 (cf. the remark at the end of Section 3.1). The resulting on-the-fly garbage collection algorithm is

\[
\text{for all} \ r \in \text{roots} \ \text{do shade (} r \text{)} ;
\text{id} := 0 \ ; \ \text{cell} := 0 \ ;
\text{repeat}
\text{if color(cell) = gray then}
\text{begin forall} \ j \in \text{children(cell)} \ \text{do shade(} j \text{)} ;
\text{id} := \text{cell} \ ; \ \text{color(cell)} := \text{black} \ \text{end} ;
\text{cell} := (\text{cell} + 1) \mod N
\text{until} \ \text{token} = N .
\]

3.3. Concurrent mutator activities

According to Section 2.2, the graph marker designed in Section 3.2 will also work when there is a concurrent mutator program, that executes the following code
indivisibly when it adds an edge \((i,j)\):

\[
\text{ADD}(i,j):
\quad \text{children}(i) \leftarrow \text{children}(i) + \{j\};
\quad \text{shade}(j).
\]

Because the DELETE action in Section 1 does not violate the correctness of the system it can remain unchanged, as argued in Section 2.2. Because we assumed (Section 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. [10].

4. A highly parallel garbage collector

As in Section 3 we will assume here that the mutator and collector processors share one array of memory cells. In this section we will 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 3.

4.1. A highly parallel termination detection protocol

The DFG protocol basically consists of sequentially visiting processes. If all processes were 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:

\[
\text{DFG}:
\begin{align*}
\text{repeat} \\
\quad & \quad \text{success} := \text{true} ; \\
\quad & \quad \text{forall processes } p \text{ do} \\
\quad & \quad \quad \text{begin wait until } p \text{ is not active} ; \\
\quad & \quad \quad \quad \text{if } p \text{ is blue then} \\
\quad & \quad \quad \quad \quad \text{begin success := false} ; p \text{ becomes idle end} \\
\quad & \quad \quad \text{end} \\
\quad & \quad \text{until success.}
\end{align*}
\]

(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:

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

Here "visit i" will consist of a termination detection visit (as in Section 3) as well as an eventual execution of MARK1.

### 4.2. 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";
- 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:

```plaintext
MARK_ROOTS:
forall i ∈ roots do shade(r).
```

A 'visit' will have 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:

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

or, more explicitly,

```plaintext
VISIT(i):
if color(i) = gray then
begin forall j ∈ children(i) do shade(j);
color(i) := black; success := false
end.
```

We will 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. 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 assign each garbage collection processor to a part. So, let \(\{S_i| 1 \leq i \leq k\}\) be a partition of \(M\). The marking
system is given by

```plaintext
forall r ∈ roots do shade(r);
repeat
  success := true;
  forall i ∈ {1, ..., k} pardo
    forall p ∈ Si do visit(p)
  until success.
See Cohen [5] 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. Therefore we introduce a 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, the round is completed. Call the two garbage collecting processes GCA and GCB. We can describe the two processes as follows:

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

GCB:
```plaintext
repeat
  synchronize;
  b := N - 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 statement 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.

4.3. Concurrent mutator actions

Concurrent mutator actions in these graph marking systems are handled as in Section 3.3.

5. 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 \( P \) of processes \( \text{MARK2}(i) \). 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.

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

Basically the protocol by Dijkstra and Scholten (hereafter called the DS protocol) is an acknowledgment 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:

\[
C(p) = \text{the number of activation messages that } p \text{ has received but not yet acknowledged,}
\]

\[
D(p) = \text{the number of activation messages that } p \text{ has sent but did not yet receive an acknowledgment for.}
\]

We call a process \( p \) engaged if \( D(p) > 0 \) 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 [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 5.1** (Safety). When \( E \) becomes unengaged, the system is terminated.

**Lemma 5.2** (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 unACKed messages. Also, when \( p \) acknowledges a message, \( p \) need not mention the message under concern.

5.2. 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 \( \text{MARK2}(i) \), a test for earlier activations is necessary anyhow.
By the primitive \textit{activate}(i, father) we will mean: send an activation message, containing the sender’s identity father, to i. The receipt of this message by \(i\) triggers execution of the following procedure, which contains the code for \textit{MARK2}(i) as well as the code for the DS protocol:

\textbf{ACTIVATE}(i, father):

\begin{verbatim}
if \(i\) was not activated before then
begin \(C(i) := 1\);
forall \(j \in \text{children}(i)\) do
\quad begin \textit{activate}(j, i); \(D(i) := D(i) + 1\) end;
\quad color(i) := black;
while \(D(i) > 0\) do
\quad begin receive an ACK; \(D(i) := D(i) - 1\) end;
end
else (* i.e., \(i\) is or has been engaged already *)
\quad signal(father).
\end{verbatim}

The primitive \textit{signal}(father) means: send an acknowledgment 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, thus ensuring mutual exclusion. State information about the process is stored explicitly. So, suppose node \(i\) contains yet another field father(i). We present the message driven form:

\textbf{ACTIVATE}(i, father): (* executed if \(i\) receives an activation message from father *)

\begin{verbatim}
if \(i\) was not activated before then
begin father(i) := father; \(C(i) := 1\);
forall \(j \in \text{children}(i)\) do
\quad begin \textit{activate}(j, i); \(D(i) := D(i) + 1\) end;
if \(D(i) = 0\) then (* \(i\) has no children *)
\quad begin color(i) := black; \(C(i) := 0\);
\quad signal(father(i)) end
end
else (* i.e., \(i\) was activated before *)
\quad signal(father).
\end{verbatim}

\textbf{SIGNAL}(i): (* executed if \(i\) receives an acknowledgement signal *)
\(D(i) := D(i) - 1;\)
\if \(D(i) = 0\) then
\quad begin color(i) := black; \(C(i) := 0\); signal(father(i)) end.
\end{verbatim}
Note, that we have deferred \textit{"color}(i) := \textit{black}" to the time of unengagement of \emph{i}. This minor change does not affect the correctness or termination properties of the algorithm. We do this, so the statements \textit{"color}(i) := \textit{black}" and \textit{"C}(i) := 0" always appear 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

- \textit{State 1: } \textit{C}(i) = 0, \textit{color}(i) = \textit{white},
- \textit{State 2: } \textit{C}(i) = 1, \textit{color}(i) = \textit{white},
- \textit{State 3: } \textit{C}(i) = 0, \textit{color}(i) = \textit{black}.

As in Section 3 wc will use a coding trick and represent \textit{C} in the \textit{color} field by using the extra color \textit{gray} to represent state 2. The test \textit{"i was not activated before"} is then replaced by \textit{"i is white"}. The program for all node processes now becomes

\begin{verbatim}
ACTIVATE(\emph{i}, \textit{father})
\begin{align*}
& \text{if } \textit{color}(\emph{i}) = \textit{white} \text{ then } \\
& \quad \text{begin } \textit{father}(\emph{i}) := \textit{father} ; \textit{color}(\emph{i}) := \textit{gray} ; \\
& \quad \quad \text{forall } \emph{j} \in \textit{children}(\emph{i}) \text{ do } \\
& \quad \quad \quad \begin{align*}
& \quad \quad \text{activate}(\emph{j}, \emph{i}) ; D(\emph{i}) := D(\emph{i}) + 1 \\
& \quad \quad \quad \text{if } D(\emph{i}) = 0 \text{ then } \\
& \quad \quad \quad \quad \text{begin } \textit{color}(\emph{i}) := \textit{black} ; \textit{signal}(\textit{father}(\emph{i})) \end{align*} \\
& \quad \text{end} \\
& \text{else} \\
& \quad \text{\textit{signal}(\textit{father}).}
\end{align*}
\end{verbatim}

\begin{verbatim}
SIGNAL(\emph{i})
\begin{align*}
& \text{D(\emph{i}) := D(\emph{i}) - 1 ;} \\
& \text{if } D(\emph{i}) = 0 \text{ then } \\
& \quad \text{begin } \textit{color}(\emph{i}) := \textit{black} ; \textit{signal}(\textit{father}(\emph{i})) \text{ end.}
\end{align*}
\end{verbatim}

The marking process is elegantly started and controlled by the following transcription of \texttt{MARK_ROOTS}:

\begin{verbatim}
E:
\begin{align*}
& (* \texttt{MARK_ROOTS} *) \\
& \text{for all } \emph{r} \in \textit{roots} \text{ do } \\
& \quad \text{begin } \texttt{activate}(\emph{r}, \textit{E}) ; D(\textit{E}) := D(\textit{E}) + 1 \text{ end ;} \\
& \text{while } D(\textit{E}) > 0 \text{ do } \\
& \quad \text{begin } \text{receive an ack ; } D(\textit{E}) := D(\textit{E}) - 1 \text{ end.}
\end{align*}
\end{verbatim}

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.
We have not given attention to the scheduling of the processes. It is assumed that the run-time system ensures that messages are eventually received and the procedures above will be executed. The algorithm can be run on an arbitrary number of processors, each with its own local memory, and allows an arbitrary 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.

5.3. 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 [12] handles it. The invariant INV', as 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 a black node, then no child of \( i \) is white.

We have B because \( i \) turns black only after receiving acknowledgments for the activation messages it sent. The mutator must not violate this invariant and thus it must activate nodes sometimes. However, in order to enable a node to send an acknowledgment 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 Lemmas 2.9 and 5.1 guarantee that such a node exists, it is not easy to find a suitable one. This is why in [12] the behavior 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: the addition of an edge \((a, c)\) by a primitive \(\text{add_grandson}(a, b, c)\), where it is assumed that edges \((a, b)\) and \((b, c)\) exist already. Obviously \(\text{add_grandson}(a, b, c)\) 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 2.

<table>
<thead>
<tr>
<th></th>
<th>white</th>
<th>gray</th>
<th>black</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>-</td>
<td>(1)</td>
<td>impossible</td>
</tr>
<tr>
<td>gray</td>
<td>-</td>
<td>-</td>
<td>(2)</td>
</tr>
<tr>
<td>black</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

*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 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, for 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.)

We can now give the primitive ADD_GRANDSON:

\[
\text{ADD\_GRANDSON}(a,b,c)
\]

\[
(* \text{a is reachable, } b \in \text{children}(a), \text{ and } c \in \text{children}(b) *)
\]

\[
\text{if color}(a) = \text{gray and color}(b) = \text{white then}
\]

\[
\begin{align*}
\text{activate}(c, a) &; \ D(a) := D(a) + 1 \\
\text{if color}(a) = \text{black and color}(b) = \text{gray then}
\end{align*}
\]

\[
\begin{align*}
\text{activate}(c, b) &; \ D(b) := D(b) + 1 \\
\text{wait until ACTIVATE}(c,b) \text{ is finished in } c &; \\
\end{align*}
\]

\[
\text{children}(a) := \text{children}(a) + \{c\}.
\]

For more primitives that can be supported in a similar way, see [12, 13].

5.4. 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 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 3.

In two cases (see Table 3) special action is needed:

\[
\begin{array}{ccc}
\text{Table 3} \\
\hline
\text{a} & \text{white} & \text{gray} & \text{black} \\
\hline
\text{white} & - & (1) & (2) \\
\text{gray} & - & - & - \\
\text{black} & - & - & - \\
\hline
\end{array}
\]

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 and 2.9 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 5.1). 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. 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.

Deletion of edges and addition of new nodes to the data structure is not dealt with here for the same reason as in Sections 3.3 and 4.3.

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

Suppose the distributed 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 5). We assume in this section that the processors form a logical ring. Accordingly the arrays of memory cells of the processors are arranged in a ring as well. 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 introduced in Section 3.1.

6.1. The graph marking system for a ring of processors

As in Section 3.2, the circulation of the token around the ring of nodes serves a dual function of scheduling and termination detection. But, we want all garbage collecting processes to do work, not just 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 the total number of memory cells, $N$, is known to every processor, and that each processor $p$ has $N_p$ cells, locally addressable as $1..N_p$. Also assume that $\text{shade}(j)$ is done by sending a 'shading message', which reaches its 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 ring, the token is actually passed as an interprocessor message. The complete marking system consists of 4 ‘modules’ ($P$ denotes any garbage collecting processor):

1. (* MARK_ROOTS *)
   
   all $P$ do
   
   for all $r \in \text{roots}$ do $\text{shade}$(r).

2. (* Local progress of marking *)
   
   all $P$ do
   
   repeat
   
   $i := \ldots$; (* a gray node in processor $P$ *)
   
   forall $j \in \text{children}(i)$ do $\text{shade}(j)$;
   
   $\text{color}(i) := \text{blue}$
   
   until $P$ is notified of termination.

3. (* Start of DFG protocol *)
   
   (* Only one processor, the leader $l$, executes: *)
   
   $\text{send}$ ($\langle 0 \rangle$) to $S(l)$.

4. (* Termination detection *)
   
   (* Upon receiving the token $\langle v \rangle$, $P$ executes: *)
   
   $\text{val} := v$
   
   for $i := 1$ to $N_p$ do
   
   begin if $\text{color}(i) = \text{gray}$ then
begin forall \( j \in \text{children}(i) \) do shade\( (j) \);
\text{color}(i) := \text{blue}
end;
if color\( (i) = \text{blue} \) then
begin \text{val} := 0 ; \text{color}(i) := \text{black} \end{align*}
\text{val} := \text{val} + 1;
if \text{val} = N \text{ then...} \text{ (*) Termination, take appropriate action *)}
end;
send (\text{val}) \text{ to } S(P).

A processor that concludes termination (in module 4) notifies all other processors. All processors will then know that the marking phase has terminated and all unmarked nodes are garbage.

6.2. Concurrent mutator actions

Here again the same concurrent mutator actions as in Section 3.3 apply.

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 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, as 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 a ‘monitor’ or ‘supervisor’ for the (basic) processes residing in that processor. The monitor 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’.

This suggests the following notions. Let \( P \) be a set of processes as in Section 1.2, and let \( P, Q, R, \ldots \) be groups (= sets) of processes, such that \( \{P, Q, R, \ldots \} \) is a partition of \( P \). 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. Assume that transmission delay within one group is always 0.
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, active, group $Q$.

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, and transmission delay of messages from members of $P$ is 0. So $y$ must be a member of some other group $Q$. □

When the situation of Lemma 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 $\{S_P, S_Q, \ldots\}$ is terminated.

7.2. A hierarchical graph marker for general networks

The ideas presented in Section 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

```plaintext
if j is white and j is 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:

```plaintext
repeat
  if queue not empty then
    begin dequeue($i$);
      forall $j \in \text{children}(i)$ do shade($j$);
      color($i$) := black
    end
  until there is global termination.
```

The processors can run any arbitrary termination detection protocol to detect the global termination of the marking phase. For termination detection protocols on arbitrary networks, see e.g. [20, 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's collector 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 timestamp $t_a$ is denoted by $\langle \text{ACT}, t_a \rangle$ and means 'this is an activation message for all $F_t, t \leq t_a$'.

Its receipt triggers the following transcription of MARK2:

\[
\{i \text{ receives } \langle \text{ACT}, t_a \rangle \} \\
\text{if } t_a > \text{timestamp}(i) \text{ then} \\
\quad \text{begin forall } j \in \text{children}(i) \text{ do} \\
\qquad \text{send } \langle \text{ACT}, t_a \rangle \text{ to } j; \\
\qquad \text{timestamp}(i) := t_a \\
\text{end.}
\]

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 $\text{redo}_P$. 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 GVT algorithms on general graphs and under more general assumptions about communication, see [21].

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

8. Conclusions

In this paper we have exploited the 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 transparent derivations of several old and new
on-the-fly garbage collecting algorithms. We have not investigated 'granularity' aspects deeply. The case studies show that the approach is useful, and apparently of sufficient generality to deal with the design of all known on-the-fly garbage collectors.

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 led to several new models of distributed computations (see Shavit and Francez [19], or Beilken et al. [2]). Hence it seems worthwhile to make a more thorough study of the mechanisms underlying distributed termination detection.

Acknowledgment

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

References


