The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes

Gerard Tel, Friedemann Mattern

RUU-CS-90-24
July 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

ISSN:0924-3275

The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes

G. Tel*
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.

F. Mattern
Department of Computer Science, Kaiserslautern University,
P.O. Box 3049, D 6750 Kaiserslautern, Fed. Rep. of Germany.

July 1990

Abstract

It is shown that the termination detection problem for distributed computations can be modeled as an instance of the garbage collection problem. Consequently, algorithms for the termination detection problem are obtained by applying transformations to garbage collection algorithms. The transformation can be applied to collectors of the "mark–and–sweep" type as well as to reference counting garbage collectors. As examples, the scheme is used to transform the weighted reference counting protocol, the distributed reference counting protocol of Lermen and Maurer, and Ben–Ari's mark–and–sweep collector into termination detection algorithms. Known termination detection algorithms as well as new variants are obtained.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.10 [**Software Engineering**]: Design—*Methodologies*; C.2.2 [**Computer–Communication Networks**]: Distributed Systems—*Network Operating Systems*; D.4.2 [**Operating Systems**]: Storage Management—*Distributed Memories.*

General Terms: Algorithms, Design, Theory, Verification.

*The work of this author was supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM).

Additional Keywords and Phrases: Distributed Algorithms, Termination Detection, Garbage Collection, Program Transformations.

1 Introduction

A large amount of the research efforts in distributed algorithm design has been devoted to the problem of detecting when a distributed computation has terminated. The reason for the large number of publications on this subject is threefold. First, as the problem has shown up under varying model assumptions, and there are several solutions for each model, a really large number of different algorithms has emerged. All of these algorithms were published separately, as unifying approaches treating a number of algorithms as a class have been rare. Second, the problem of termination detection, being sufficiently easy to define and yet non–trivial, has been seen as a good candidate to illustrate the merits of design or proof methods for distributed algorithms. Third, it has been observed that the fundamental difficulties of the termination detection problem are the same as those of other problems in distributed computing. Termination detection algorithms are related to algorithms for computing distributed snapshots [CL85], detecting deadlocks [CMH83, Na86], and approximating a distributed infimum [ST88]. Thus the problem is seen to be important from both a practical, algorithmical, as well as from a theoretical, methodological point of view.

From both points of view we consider it useful to recognize general design paradigms for distributed termination detection algorithms. One such paradigm was described in [Te86]. A new paradigm is presented in this paper. It is shown that the semantics of the termination detection problem is fully contained in the semantics of the garbage collection problem. As a result, termination detection algorithms are obtained as suitable instantiations of garbage collection algorithms.

Subsection 1.1 introduces the termination detection problem. Subsection 1.2 introduces the (distributed) garbage collection problem. Section 2 describes how the termination detection problem can be formulated as garbage collecting one hypothetical object and derives the algorithmical transformation. Section 3 applies the transformation to three known garbage collection algorithms. Section 4 contains some additional remarks and comments.

1.1 The Termination Detection Problem

In a distributed system where processes communicate only via messages, in general no process has a consistent and up to date view of the global state. As a result, it is non–trivial to decide whether or not the global state is one in which a distributed computation has terminated. Some processes may have finished their local computations, while others are still executing. But tasks may migrate from one process to another, new tasks may be generated, or the receipt of a message may result in

renewed computational activity. As a consequence, finished processes may later be computing again.

In general it is not possible for a process to decide whether new tasks will later be generated by the process. Therefore it is always assumed that for each process a local *condition of stability* is defined. While this local condition holds, the process does not send messages (belonging to the computation) to other processes, no new tasks are generated by the process, and no initiative of the process itself falsifies the condition of stability. Only the receipt of a message (belonging to the computation) can do so. It now follows that if a global state is reached in which all processes (simultaneously) satisfy their condition of stability and no messages are in transit, the computation is terminated because the system as described will forever remain as dead as a doornail. A control computation must be superimposed to detect this situation.

1.1.1 Description of the Problem

The problem is described formally as follows. A collection **P** of *processes* is considered, communicating by message passing. A process is either *passive* (if its condition of stability is satisfied) or *active* (if the condition is not satisfied). *Active* processes may send messages, but *passive* processes don't. An *active* process may spontaneously become *passive*, but a *passive* process may become *active* only on receipt of a message.

A full description of the possible actions of processes is given below. Throughout the paper it is assumed that each action is executed atomically. An action whose name is subscripted with p takes place in process (or object) p. An assertion between braces ("{" and "}") is a guard and means that the action can only be executed when the assertion is true. Comments are placed between "(*" and "*)". Action $\mathbf{S}_p$ is the sending of a message, action $\mathbf{R}_p$ the receipt of a message, and action $\mathbf{I}_p$ the transition of a process from *active* to *passive*.

$\mathbf{S}_p$: { $state_p = active$ }
send a message M

$\mathbf{R}_p$: (* A basic message arrives *)
receive message M ; $state_p := active$

$\mathbf{I}_p$: (* The process becomes *passive* *)
$state_p := passive$

Define the *termination condition* as:

No process is *active* and no messages are in transit.

This condition is stable: once true, it remains so. It is required to superimpose on the described *basic* computation a *control* computation which enables one or more

of the processes to detect when the termination condition holds. A process detects this by entering a special state *terminated.* The following two criteria specify the correctness of the control algorithm.

D1 *Safety.* If any process is in state *terminated* then the termination condition holds.

D2 *Liveness.* If the termination condition holds, then eventually a process will be in the *terminated* state.

A *passive* process may take part in this control computation, and receiving control messages does not make a *passive* process *active.*

Under varying assumptions about the communication semantics the concise description above still allows different variants of the problem. Originally the problem emerged from a CSP context, where the sending and receipt of a message are synchronized with each other. Thus messages are never in transit; the *termination condition for synchronous communication* simply reads "all processes are passive". The introduction of asynchronous communication complicates the problem, as somehow it must be verified that the channels are empty. This can be done using special *marker* messages (in a FIFO environment) [Mi83], acknowledgements [DS80], or counting of sent and received messages [Ma87].

1.1.2 Solutions to the Problem

Several classes of solutions to the termination detection problem are known. The most important ones are those based on *probes* and those based on *acknowledgements.*

Probe–based algorithms. A *probe* is a distributed algorithm that "visits" all processes in the network. (It can be implemented by a token circulating on a ring, by an echo mechanism, or in many other ways [Te90].) To detect termination using probes, it is attempted to maintain that all visited processes are *passive*, and no message is underway to or from a visited process. A violation of this aim occurs when a non–visited process sends a message to a visited one. Usually, if this happens the current probe is marked as unsuccessful, and after its completion a new probe is initiated. (This marking can be done, for example, by assuming a different "color" for processes that caused the violation and probe messages that report about it.) The most notorious example in this class is [DFG83], a general treatment is given in [Te86].

Acknowledgement–based solutions. In these algorithms all messages of the basic computation are acknowledged, but only after all computational activity resulting from it has ceased. That is, if an *active* process receives a message, it acknowledges it immediately. If a *passive* process receives a message and becomes

active, it defers the acknowledgement until it is *passive* again, and has received acknowledgements for all messages it sent during the period of activity. Examples of this class are [DS80, SF86, CV90].

1.2 The Distributed Garbage Collection Problem

As our approach for deriving termination detection algorithms is based on solutions to the garbage collection problem, we shall now describe this problem. From a practical point of view, algorithms for the garbage collection problem are important for the storage management of programming languages with dynamic objects. They are also used in the implementation of functional programming languages as these languages operate on directed graphs, represented by memory cells referencing each other through pointers. An account of various garbage collection algorithms for multiprocessors and distributed systems may be found in [Ru88].

In many papers different models for the problem are found, here a model based on the communicating objects paradigm is presented which is close to the model of Lermen and Maurer [LM86]. The advantage of this model is that it abstracts from aspects which are not relevant to our purposes, such as processors, memory cells, and the difference between "local" and "remote" references.

1.2.1 Description of the Problem

An (object–oriented) distributed system consists of a collection **O** of cooperating processes called *objects*. A subset of **O** is designated as *root objects*. Objects are able to hold *references* to other objects. These references can be transmitted in messages, see below. A reference to an object r will be called an r–reference. An object r is a *descendant* of q if q holds an r–reference or a message containing an r–reference is in transit to q. An object is *reachable* if it is a root object or a descendant of a reachable object. An object p holding an r–reference may decide to *delete* it, after which p no longer holds this reference. Also, a reachable object p holding an r–reference may *copy* the reference to another object q, after which q will hold an r–reference, by sending the r–reference in a message to q. Object q will hold the r–reference after receipt of this message. An object can have multiple references to the same target object. Formally, the allowed actions in this model are as follows.

$\mathbf{C}_p$: (* p copies an r–reference to q *)
{ p is reachable and holds an r–reference }
send a copy(r) message to q

$\mathbf{R}_p$: (* A copy(r) message arrives *)
receive the copy(r) message ;
insert the r–reference

D_p: (* p deletes an r–reference *)
{ p holds an r–reference }
delete the r–reference

An object is called *garbage* if it is not reachable. As only references to reachable objects are copied, a garbage object remains garbage forever (i.e., being garbage is a stable property of an object). For reasons of memory management it is required that garbage objects are identified and collected. This task is taken care of by a garbage collecting algorithm. The following two criteria define the correctness of a garbage collecting algorithm.

G1 *Safety.* If an object is collected, it is garbage.

G2 *Liveness.* If an object is garbage, it will eventually be collected.

1.2.2 Solutions to the Problem

Many solutions have been proposed to the distributed garbage collection problem, most of which fall into one of two categories: collectors of the *reference counting* type and collectors of the *mark–and–sweep* type. Both types of solutions are known since over 30 years for classical, non–distributed systems [Co60, McC60].

Reference counting [LM86, Be89, WW87]. Collectors of the first type maintain for each object a count of the number of references in existence to that object. References in other objects as well as references in messages are taken into account. This reference count is incremented when the reference is copied, and decremented when the reference is deleted. When the count for a non–root object drops to zero, it can be concluded that the object is garbage and consequently the object can be collected. For root objects no reference count is maintained.

A group of garbage objects, cyclically referencing each other, cannot be collected by a reference counting algorithm, because no reference count drops to zero. Thus the algorithms do not satisfy the liveness condition, and usually a supplementary algorithm (typically of the mark–and–sweep type) is used to collect cyclic structures of garbage. In our application, however, cyclic structures of garbage objects do not occur, and a supplementary algorithm is not necessary.

Mark–and–sweep [St75, Be84, Dij78]. Collectors of the second type mark all reachable objects as such, starting from the roots and recursively marking all descendants of marked objects. In this way all reachable objects become marked eventually. The design of the marking algorithm is complicated by the possibility that references are inserted and deleted during its operation. The objects in the system must cooperate with the marking algorithm by also marking objects when references are made or changed. A possible design consists of an algorithm for the marking proper, upon which a termination detection algorithm is superimposed

[TTL88]. When the marking phase is terminated a sweep through all objects is made, in which all unmarked objects are collected. These two phases form one cycle of the collector, and cycles are repeated as long as necessary.

2 Termination Detection Using Garbage Collection

In this section we describe how the distributed termination detection problem in general can be modeled as an instance of the garbage collection problem. As a result, solutions to the termination detection problem can be derived from garbage collection algorithms, of which examples will be shown in section 3. First the collection **O** of objects used for this purpose is described as well as the behavior of these objects. Next it is shown that the termination condition is equivalent to one particular object becoming garbage. As a result, termination can be detected by a garbage collection algorithm.

Recall that **P** is the set of processes whose termination is to be detected. The collection **O** of objects consists of one root object A_p for every process p in **P**, and a single *indicator object* Z. An object A_p may send and receive the messages of the basic computation, and has all the variables p has. It is called *passive* (*active*) when process p is *passive* (*active*). As A_p is a root object, it is always reachable.

The indicator object Z is not a root object. Its only purpose is to indicate the termination condition with its reachability status by the following equivalence, which will be maintained during execution.

$$Z \text{ is garbage} \Leftrightarrow \text{the termination condition holds.} \qquad \text{(IND)}$$

Theorem 2.1 *IND holds when the following two rules are observed:*

R1 An active object holds a Z–reference. A passive object holds no Z–reference.

R2 Each message of the basic computation contains a Z–reference.

Proof. Z is garbage is equivalent to: Z is not a descendant of any of the A_p. By definition, this means that no A_p holds a Z–reference, and to no A_p a message is in transit containing a Z–reference. By R1 and R2 this is equivalent to: no A_p is *active* and to no A_p a message (of the basic computation) is in transit. This is the definition of the termination condition. □

It must be shown that R1 and R2 can be maintained. It is possible to ensure through proper initialization that R1 and R2 hold initially. To this end, assume that *active* objects are initialized with the necessary Z–reference, and *passive* objects without it, and that messages in transit initially contain the reference also. To maintain R1 and R2 during the distributed computation, each transmission of a message copies the Z–reference, and processes delete the reference when they become *passive*. More explicitly, the actions to be carried out by A_p are modified as follows:

$\mathbf{S}_p$: { $state_p = active$ }
send a message $\langle M, Z \rangle$

$\mathbf{R}_p$: (* A basic message arrives *)
receive message $\langle M, Z \rangle$; $state_p := active$;
insert Z in the references of A_p

$\mathbf{I}_p$: (* The process becomes *passive* *)
$state_p := passive$;
delete Z from the references of A_p

With these modifications R1 and R2 are maintained indeed. R1 is maintained because Z–references are deleted in action $\mathbf{I}_p$, and inserted in action $\mathbf{R}_p$. The latter is possible because the message contains a Z–reference by R2. R2 is maintained because in action $\mathbf{S}_p$ a Z-reference is included in every message. This is possible because only *active* objects send messages, and these contain a Z–reference by R1. Thus R1 and R2 are maintained during computation, and by theorem 2.1 IND holds. To arrive at a termination detection algorithm, superimpose upon the objects as described a garbage collection algorithm. The garbage collection algorithm is then modified so as to inform the objects A_p when it collects Z. When receiving this notice, the root objects enter the *terminated* state. We omit this (trivial) operation from the description of the algorithms that are to follow.

Theorem 2.2 *The algorithm as constructed satisfies conditions D1 and D2.*

Proof. Assume any process enters the *terminated* state. This happens upon notice that Z is collected. By the correctness of the garbage collection algorithm (condition G1) this implies that Z is garbage. By theorem 2.1 the termination condition holds.

Assume the termination condition holds. By theorem 2.1, Z is garbage, hence, by the liveness of the garbage collector (condition G2) Z will eventually be collected. Notice of this will be sent to the processes, and these will enter the *terminated* state in finite time. □

It was remarked in section 1.2.2 that garbage collectors of the reference counting type are not able to collect cyclic structures of garbage, which may possibly harm the liveness of the termination detection algorithm. It is however easily seen that Z is not part of such a cyclic structure, and in fact the following, stronger equivalence holds.

The termination condition holds $\Leftrightarrow$ there are no references to Z.

Summary of the transformation. The construction of a termination detection algorithm is summarized in the following four steps.

1. Form the set **O** of objects, consisting of the root objects A_p and one indicator object Z.

2. Superimpose upon the actions of the basic computation the handling of the Z–reference.

3. Superimpose upon this combined algorithm a garbage collection algorithm.

4. Replace the collection of Z by a notification of termination.

3 Examples of the Transformation

The transformation described in section 2 can in principle be applied to any garbage collection scheme, of the reference counting as well as the mark–and–sweep type or working according to other principles. In the next two subsections "simple" reference counting and weighted reference counting are considered and corresponding termination detection algorithms are derived. In subsection 3.3 the transformation is applied to a mark–and–sweep garbage collector.

3.1 Reference Counting

This subsection transforms the distributed reference counting algorithm of Lermen and Maurer [LM86] into a termination detection algorithm. For each non–root object o a reference count RC_o is maintained. When an o–reference is copied from object p to object q, p sends to q a copy message $\mathbf{cop}(o)$ and to o an increment message $\mathbf{inc}(o, q)$. When an o–reference is deleted by p, p sends a delete message (or *decrement* message) $\mathbf{dec}(o)$ to o.

The scheme is much complicated by the possibility that $\mathbf{dec}(o)$ and $\mathbf{inc}(o, q)$ messages may arrive at o in a different order than they are sent. This is to be expected when message communication is not FIFO or when these messages are sent from different objects. If an $\mathbf{inc}(o, q)$ message is overtaken by a $\mathbf{dec}(o)$ message, RC_o may temporarily drop to 0, causing o to be collected while it is reachable. Lermen and Maurer overcome this problem by introducing a synchronization discipline between $\mathbf{dec}(o)$ and $\mathbf{inc}(o, q)$ messages.

3.1.1 Description of the Scheme

The necessary synchronization is achieved by a two–way strategy. First, object o learns about the creation of a certain o–reference *before* it learns about the deletion of this o–reference. Second, o learns about the creation of all *copies* of a certain o–reference before it learns about the deletion of the original reference. In [LM86] it is shown that this indeed implies the safety of the scheme.

To implement the first part, o sends to q an acknowledgement **ack**(o) for the message **inc**(o, q) it receives from p. Note that the communication scheme is triangular: p sends **cop**(o) to q and **inc**(o, q) to o, o sends **ack**(o) to q, and q receives both a **cop**(o) message (from p) and an **ack**(o) message (from o). The **ack**(o) message informs q that o has learned about the creation of its o–reference. q deletes an o–reference only if it is an acknowledged reference, that is, q has received an **ack**(o) message for it. q maintains a count both of the number of its acknowledged references to o and of the number of "surplus" copy messages it has received.

To implement the second part, a FIFO discipline on the links is assumed. As p sends **inc**(o, q) messages concerning copies of its o–reference earlier than the **dec**(o) message, this ensures indeed that o is informed about the creation of copies before it learns about the deletion of the reference.

A formal description of the actions in the scheme is given below. The actions to create a new object are omitted, because creation of objects does not occur when the scheme is used for termination detection. Each object p keeps, besides its bag of references, the following two variables for each non–root object o. $aR_p(o)$ is the number of acknowledged o–references. $iR_p(o)$ is the difference between the number of **cop**(o) messages and the number of **ack**(o) messages received by p. Initially there are only acknowledged references ($iR_p(o) = 0$ for all p, o), the reference counts correctly reflect their number ($RC_o = \sum_p aR_p(o)$), and no messages are in transit. The following actions are possible.

CR$_p$: (* Copy an o–reference to q *)
send **cop**(o) to q ; send **inc**(o, q) to o

DR$_p$: (* Delete an (acknowledged) o–reference *)
{ $aR_p(o) > 0$ }
delete the o–reference ;
send **dec**(o) to o ; $aR_p(o) := aR_p(o) - 1$

RI$_o$: (* An **inc**(o, q) message arrives at o *)
receive **inc**(o, q) ;
$RC_o := RC_o + 1$; send **ack**(o) to q

RD$_o$: (* A **dec**(o) message arrives at o *)
receive **dec**(o) ; $RC_o := RC_o - 1$;
if $RC_o = 0$ **then** collect o

RA$_p$: (* An **ack**(o) message arrives at p *)
receive **ack**(o) ;
if $iR_p(o) > 0$ **then** $aR_p(o) := aR_p(o) + 1$;
$iR_p(o) := iR_p(o) - 1$

$\mathbf{RC}_p$: (* A $\mathbf{cop}(o)$ message arrives at p *)
receive $\mathbf{cop}(o)$; insert the o–reference ;
if $iR_p(o) < 0$ **then** $aR_p(o) := aR_p(o) + 1$;
$iR_p(o) := iR_p(o) + 1$

A full correctness proof for this garbage collection scheme is given in [LM86]. Under the rules for the computation of the objects, an object already holding an o–reference may receive yet another $\mathbf{cop}(o)$ message. In order to send enough $\mathbf{dec}(o)$ messages in such a case, the action $\mathbf{DR}_p$ is executed sufficiently often to make $aR_p(o)$ equal to 0. An $\mathbf{ack}(o)$ message may even arrive when the reference is no longer needed, in which case the execution of $\mathbf{DR}_p$ is triggered by the receipt of the $\mathbf{ack}(o)$ message (if $\mathbf{RA}_p$ results in $aR_p(o) > 0$). The termination detection algorithm, obtained from this scheme in the next subsection, must also allow multiple execution of the corresponding action (see also the remarks at the end of section 3.1.2).

3.1.2 Transformation into a Termination Detection Algorithm

In this subsection a termination detection algorithm is derived from the Lermen–Maurer scheme. A discussion of the properties of the derived algorithm, called the *Activity Counting* algorithm, is deferred to subsection 3.1.3. The termination detection algorithm is derived in the four steps described at the end of section 2.

1. The set $\mathbf{O}$ of objects consists of the objects A_p and the indicator object Z.

2. Superimpose upon the actions of the basic computation the handling of the Z–reference. This yields the following program text.

   $\mathbf{S}_p$: { $state_p = active$ }
   send a message $\langle M, Z \rangle$

   $\mathbf{R}_p$: (* A basic message arrives *)
   receive message $\langle M, Z \rangle$; $state_p := active$;
   insert Z in the references of A_p

   $\mathbf{I}_p$: (* The process becomes *passive* *)
   $state_p := passive$;
   delete Z from the references of A_p

3. Superimpose upon this combined algorithm the garbage collection algorithm of Lermen and Maurer. To this end, the $\mathbf{CR}_p$ action is included in the $\mathbf{S}_p$ action, the $\mathbf{RC}_p$ action is included in the $\mathbf{R}_p$ action, and the $\mathbf{DR}_p$ action is included in the $\mathbf{I}_p$ action. In all cases Z is substituted for o.

   $\mathbf{S}_p$: { $state_p = active$ } (* This implies p holds a Z–reference *)
   send a message $\langle M, \mathbf{cop}(Z) \rangle$ to q ; send $\mathbf{inc}(Z, q)$ to Z

$\mathbf{R}_p$: (* A basic message arrives *)
receive message $\langle M, \mathbf{cop}(Z) \rangle$; $state_p := active$;
insert Z in the references of A_p ;
if $iR_p(Z) < 0$ **then** $aR_p(Z) := aR_p(Z) + 1$;
$iR_p(Z) := iR_p(Z) + 1$

$\mathbf{I}_p$: (* The process becomes *passive* *)
{ $aR_p(Z) > 0$ }
$state_p := passive$;
delete Z from the references of A_p ;
send $\mathbf{dec}(Z)$ to Z ; $aR_p(Z) := aR_p(Z) - 1$

$\mathbf{RI}_Z$: (* An $\mathbf{inc}(Z, q)$ message arrives at Z *)
receive $\mathbf{inc}(Z, q)$;
$RC_Z := RC_Z + 1$; send $\mathbf{ack}(Z)$ to q

$\mathbf{RD}_Z$: (* A $\mathbf{dec}(Z)$ message arrives at Z *)
receive $\mathbf{dec}(Z)$; $RC_Z := RC_Z - 1$;
if $RC_Z = 0$ **then** collect Z

$\mathbf{RA}_p$: (* An $\mathbf{ack}(Z)$ message arrives at p *)
receive $\mathbf{ack}(Z)$;
if $iR_p(Z) > 0$ **then** $aR_p(Z) := aR_p(Z) + 1$;
$iR_p(Z) := iR_p(Z) - 1$

4. Replace the collection of Z by a notification of termination. This is done by substituting

   send **term** to all A_p

   for "collect Z" in action $\mathbf{RD}_Z$. Upon receipt of this message, the processes enter the *terminated* state.

The derivation of the termination detection algorithm is now completed. Finally some simplifications in the algorithm can be made. Because there is only one non–root object, the subscript Z may be dropped from all variables. Furthermore, the handling of the Z–reference only serves to lead the garbage collection scheme in its actions. Now that these actions have been correctly connected to the actions of the basic distributed computation this reference handling can be removed. The resulting Activity Counting algorithm is completely described as follows.

$\mathbf{S}_p$: { $state_p = active$ }
send a message M to q ; send $\mathbf{inc}(q)$ to Z

$\mathbf{R}_p$: (* A basic message arrives *)
receive message M ; $state_p := active$;
if $iR_p < 0$ **then** $aR_p := aR_p + 1$;
$iR_p := iR_p + 1$

$\mathbf{I}_p$: (* The process becomes *passive* *)
{ $aR_p > 0$ }
$state_p := passive$;
send **dec** to Z ; $aR_p := aR_p - 1$

$\mathbf{RI}_Z$: (* An **inc**(q) message arrives at Z *)
receive **inc**(q) ;
$RC := RC + 1$; send **ack** to q

$\mathbf{RD}_Z$: (* A **dec** message arrives at Z *)
receive **dec** ; $RC := RC - 1$;
if $RC = 0$ **then** send **term** to all A_p

$\mathbf{RA}_p$: (* An **ack** message arrives at p *)
receive **ack** ;
if $iR_p > 0$ **then** $aR_p := aR_p + 1$;
$iR_p := iR_p - 1$

The initial conditions for this algorithm are: $iR_p = 0$ for all p; $aR_p = 0$ if p is *passive*, and $aR_p > 0$ if p is *active*; and $RC = \sum_p aR_p$.

A process may receive an activation message when it is already *active*, in which case it does not become "even more *active*". In order to send enough **dec** messages, the process must later execute action $\mathbf{I}_p$ as many times as it has received activation messages. This is taken care of by the guard of this action, based on the proper administration of the **ack** and M messages received (variables aR_p and iR_p). When the process is *passive*, but has positive aR_p, it eventually executes $\mathbf{I}_p$ and sends a **dec** message.

3.1.3 Discussion of the Algorithm

We do not provide a full correctness proof of the Activity Counting algorithm. According to theorem 2.2 the correctness of the derived algorithm follows from the correctness of the Lermen–Maurer reference counting scheme, which was proved in [LM86].

The principle of the Activity Counting algorithm is simple. When a process activates another process, it informs the central controller Z by sending an increment message **inc**(q), and when a process becomes *passive*, it informs Z by sending a

decrement message **dec**. The controller tries to keep an account of the number of "currently" *active* processes by counting the increment and decrement messages. When enough decrement messages have been received to balance the increment messages and the initially *active* processes, it signals termination. Unfortunately the possible delay between a basic action and its registration by Z render this over–simplified scheme incorrect as the following example shows.

1. Assume only p is *active* and $RC = 1$.
2. p sends an activation message to q and an increment message to Z.
3. q receives the activation message and becomes *active*. Then q becomes *passive* again and sends a decrement message to Z.
4. Z receives the decrement message (before the increment message), and RC drops to 0 while p is still *active*.

The Activity Counting algorithm takes care of this and similar scenarios, because the sending of the decrement message is deferred until an acknowledgement message **ack** has been received. This implies that Z receives the decrement message *after* it has received the corresponding increment message.

As the Activity Counting algorithm was derived from the Lermen–Maurer reference counting scheme, it inherits properties of the latter algorithm, which will now be discussed.

1. Message Complexity. The message overhead of the new algorithm is considerable: for each basic message of the computation the algorithm adds up to *three* control messages (**inc**(q), **ack**, and **dec**). A known worst case lower bound for this overhead is *one* control message per basic message [CM86], and this bound is achieved by the algorithm in [DS80].

2. FIFO Discipline on Links. For the correctness of the algorithm it is required that links deliver messages in the order they were sent.

3. Central Controller. The central object Z acts as a central controller in the Activity Counting algorithm. For *each* single transmission of a basic message up to *two* actions are necessary in Z ($\mathbf{RI}_Z$ and $\mathbf{RD}_Z$). The central process may become a bottleneck in the computation and slow down the operation of the entire system. It is not obvious how the functionality of Z can be distributed in a consistent manner.

4. Inhibition. A control algorithm is said to be *inhibitory* if it may temporarily disable actions of the basic computation. In the Lermen–Maurer scheme this is the case for the delete action, which is deferred until the object has an acknowledged version of the reference. As a consequence, in the Activity Counting algorithm

becoming *passive* is only allowed if $aR_p(o) > 0$, thus formally the algorithm is inhibitory. This disadvantage can be overcome by a slight modification of the algorithm as follows. The object may delete any reference, but the **dec**(o) message is held back if the acknowledgement for the reference was not yet received. A similar modification makes the Activity Counting algorithm non–inhibitory.

3.1.4 Related Algorithms

The Vector Counting Algorithm. With the help of **ack** messages, the Activity Counting algorithm guarantees that Z only takes into account the receipt of a basic message when it has already considered the corresponding sending of that message. Thus, Z always has a *consistent view* of the message counters. By keeping more information in Z, however, it is also possible to achieve this without actually sending **ack** messages. (The FIFO property, however, is still necessary.) For this purpose Z keeps a vector (i.e., an integer array) V with one component for each process. Whenever Z receives an **inc**(q) message it increments q's component of V instead of sending an **ack** message: $V[q] := V[q] + 1$. Action $\mathbf{RA}_p$ and the variables iR_p are no longer necessary: a process increments aR_p when it receives a basic message and sends aR_p **dec** messages to Z immediately when it becomes passive. When Z receives a **dec** message from q it decrements the corresponding component of V: $V[q] := V[q] - 1$. Notice that temporarily $V[q]$ might become negative—this is the case if Z receives a **dec** message *before* it receives the corresponding **inc**(q) message. This is precisely the situation which is avoided by use of **ack** messages in the original Activity Counting algorithm. Because Z's view is inconsistent when a component of V is negative, nothing is deduced in that case. It follows that Z can signal termination when V becomes the null vector, and the RC counter is no longer necessary. Some further optimizations (e.g., batching **inc**(q) and **dec** messages) yield a centralized variant of the so-called *Vector Counter* termination detection algorithm [Ma87, ST88]. This algorithm has lower message overhead than the Activity Counting algorithm, does not rely on the FIFO property, and can easily be realized in a distributed way as well.

Variations of the Lermen–Maurer Scheme. Two variants of the Lermen–Maurer scheme were proposed by Rudalics [Ru88]. We describe informally his *three message protocol*, which does not rely on FIFO links. In the Lermen–Maurer scheme object q receives an **ack**(o) acknowledgement when an o–reference has been copied to it, but in the three message protocol an object p receives an acknowledgement when it has initiated a copy of an o–reference. A delete message for the o–reference may be send only when all acknowledgements have been received, and to this end an acknowledgement counter is added to each reference. The protocol works as follows.

1. To copy an o–reference to q, p increments the acknowledgement count of its reference and sends an increment message to o.

2. On receipt of this message, o increments its reference count and sends a copy message to q.

3. On receipt of the copy message, q inserts the reference and sends an acknowledgement to p.

4. On receipt of this acknowledgement, p decrements the acknowledgement count of the o–reference.

When p deletes the reference, it holds back the delete message until all acknowledgements have been received. To this end, the references are all installed with acknowledgement count equal to 1, and deletion of the reference is done by decrementing the count. When the count drops to zero (either by deletion of the reference or by receipt of an acknowledgement) the delete message is sent.

A drawback of this algorithm is that the new reference can only be installed after two messages have been propagated (one from p to o and one from o to q). In Rudalics' *four message protocol* p also sends a copy message to q directly, and q installs the reference when it receives a copy message either from p or from o. The acknowledgement is sent when both copy messages have been received.

In a similar way as for the Lermen–Maurer scheme a termination detection algorithm can be derived from the three and four message protocols by the transformation of section 2.

3.2 Weighted Reference Counting

In this section we consider the transformation of a garbage collection algorithm based on *weighted* reference counting. The resulting termination detection algorithm turns out to be an already known algorithm: it was proposed in [Ma89].

As mentioned in the introduction of section 3.1, $\mathbf{dec}(o)$ and $\mathbf{inc}(o, q)$ messages in the "simple" reference counting scheme must be synchronized because their reordering may render the scheme unsafe. This need for synchronization can be avoided using weighted reference counting. In this variant each reference has a positive *weight*. The reference count of an object now represents the total weight of the references pointing to it rather than their number. (We continue to use the word reference *count* although it may no longer be completely appropriate.) When a reference is copied, its weight is *split* among the existing and the new reference. Thus, although the *number* of references increases, the *weight* remains the same, and the reference count need not be incremented and no $\mathbf{inc}(o, q)$ message need be sent. The reference count monotonically decreases (because delete messages return the weight), and the order in which control messages (only delete messages) arrive at the object becomes irrelevant.

3.2.1 Description of the Scheme

Distributed weighted reference counting schemes have been given by Bevan [Be89], Watson and Watson [WW87], and others. The principle was attributed to Weng [We79]. In the description below again the mechanism to create new objects is omitted. An o–reference is now a tuple (o, w), where w denotes the weight of the reference. Initially for each non–root object o, the reference count RC_o equals the sum of the weights of all existing o–references. The following (atomic) actions can take place.

$\mathbf{CR}_p$: (* p copies reference (o, w) to q *)
send $\mathbf{cop}(o, w/2)$ to q ; $w := w/2$

$\mathbf{RR}_p$: (* A message $\mathbf{cop}(o, w)$ arrives *)
receive $\mathbf{cop}(o, w)$;
if p has an o–reference
then add w to its weight
else insert the o–reference with weight w

$\mathbf{DR}_p$: (* p deletes reference (o, w) *)
send $\mathbf{dec}(o, w)$ to o ; delete the o–reference

$\mathbf{RD}_o$: (* A $\mathbf{dec}(o, w)$ message arrives at o *)
receive $\mathbf{dec}(o, w)$; $RC_o := RC_o - w$;
if $RC_o = 0$ **then** collect o

A correctness proof and analysis of the scheme is found in [Be89] or [WW87] and is based on invariance of the following two assertions:

1. Each reference has a positive weight; each delete message contains a positive weight.
2. $RC_o = \sum_{R=(o,w)} w + \sum_{D=\mathbf{dec}(o,w)} w$, where R ranges over all o–references in existence and D ranges over all delete messages in transit.

Action $\mathbf{RR}_p$ above adds the weight of the received reference if p already has an o–reference. There are two alternatives. First, p may return the received weight to o immediately in a $\mathbf{dec}(o, w)$ message. Second, p may store the weight separately and thus keep a non–empty *set* of weights for each reference rather than a single weight. Both alternatives maintain the two invariants of the algorithm. A consequence of the two alternative strategies is, that all weights in the system are always (negative) powers of 2, and can be represented concisely by their negative logarithm. The version above allows for a lower message complexity, as weights are combined and fewer delete messages may be necessary.

3.2.2 Transformation into a Termination Detection Algorithm

To transform the garbage collection scheme into a termination detection algorithm we apply the four step construction of section 2. Steps 1, 2, and 4 are as in section 3.1.2. In step 3 the actions of the weighted reference counting scheme are superimposed on the program resulting from step 2 (see section 3.1.2). To this end, action $\mathbf{CR}_p$ is included in action $\mathbf{S}_p$, action $\mathbf{RR}_p$ is included in action $\mathbf{R}_p$, and action $\mathbf{DR}_p$ is included in action $\mathbf{I}_p$. Again for o the object Z is substituted. This results in the following program text.

$\mathbf{S}_p$: { $state_p = active$ } (* Thus p has a Z–reference (Z, w) *)
send a message $\langle M, \mathbf{cop}(Z, w/2)\rangle$; $w := w/2$

$\mathbf{R}_p$: (* A basic message arrives *)
receive message $\langle M, \mathbf{cop}(Z, w)\rangle$; $state_p := active$;
if p has a Z–reference
then add w to its weight
else insert the Z–reference with weight w

$\mathbf{I}_p$: (* The process becomes *passive* *)
$state_p := passive$;
delete the Z–reference ; send $\mathbf{dec}(Z, w)$ to Z

$\mathbf{RD}_Z$: (* A $\mathbf{dec}(Z, w)$ message arrives at Z *)
receive $\mathbf{dec}(Z, w)$; $RC_Z := RC_Z - w$;
if $RC_Z = 0$ **then** collect Z

The same simplifications as in section 3.1.2 can be made: The actual handling of the Z reference can be removed, instead we equip every process p with a variable W_p, representing the weight of p's (virtual) Z–reference (0 if p has no such reference). The subscript Z is dropped. This finally results in the following algorithm.

$\mathbf{S}_p$: { $state_p = active$ } (* Thus $W_p > 0$ *)
send a message $\langle M, W_p/2\rangle$; $W_p := W_p/2$

$\mathbf{R}_p$: (* A basic message arrives *)
receive message $\langle M, W\rangle$; $state_p := active$;
$W_p := W_p + W$

$\mathbf{I}_p$: (* The process becomes *passive* *)
$state_p := passive$;
send $\mathbf{dec}(W_p)$ to Z ; $W_p := 0$

RD: (* A $\mathbf{dec}(W)$ message arrives at Z *)

receive **dec**(W) ; $RC := RC - W$;
if $RC = 0$ **then** send **term** to all A_p

The initial conditions for this algorithm are: $W_p = 0$ if p is *passive*; $W_p > 0$ if p is *active*; $RC = \sum_p W_p$; and no messages are in transit.

3.2.3 Discussion of the Algorithm

The termination detection algorithm that has just been derived is known as the *Credit Recovery* algorithm [Ma89].

Weight Underflow. The implementation of the weighted reference counting scheme faces a difficulty that has not yet been discussed, and it is not surprising that the Credit Recovery algorithm faces a similar difficulty. The problem arises because weights are represented in a finite number of bits: thus there is a smallest possible positive weight, and if a reference of this weight is copied its weight cannot be split. The problem in the Credit Recovery algorithm arises when a process with the smallest possible positive value of W_p sends a message.

Also the solutions to these difficulties are similar in the two algorithms. In the weighted reference counting algorithms, a new *indirection object* is created with a *maximal* reference count, and the original reference is replaced by a reference to the indirection object, with maximal weight. Next it can be copied without difficulties. In the Credit Recovery algorithm, a process negotiates with Z to exchange its (minimal) credit for a new, maximal credit. Then it can send the message. The operation results in an increase in the reference count of Z.

These additions to the algorithms do not make them inefficient or impractical, because in both algorithms weight underflow is supposed to be a very rare event. As remarked at the end of section 3.2.1, it can be arranged that all weights are powers of 2, and can be represented by their (negative) logarithm. When copying a reference, $W := W + 1$ is executed instead of $W := W/2$, and the probability of overflow in W should not be much greater than the overflow probability in classical reference counting schemes.

Generational Reference Counting. An alternative reference counting scheme, called *Generational* reference counting, has been proposed by Goldberg [Go89]. This scheme avoids **inc**(o, q) messages by sending together with a **dec**(o) message for any reference, the number of copies that have been made of this reference. Each reference has a *generation* number (a copy of a reference with generation number i gets generation number $i + 1$) and an object keeps separate counts for each generation. The communication pattern is the same as for the weighted reference counting scheme: a control message is sent only upon deletion of a reference. We omit a full description of the scheme and the (straight–forward) transformation into a termination detection algorithm.

The resulting *Generational Termination Detection algorithm* is presented below. Each *active* process p has a *generation number* gen_p and a counter $sons_p$ to count activation messages sent. Both are returned to Z when p becomes *passive*. The central controller Z maintains an array of integers *ActCount*. Initially a process p is either *active* with $gen_p = 1$ or *passive*, *ActCount*[1] equals the number of *active* processes, $ActCount[i] = 0$ for $i > 1$, and no messages are underway.

S$_p$: { $state_p = active$ }
send $(M, gen_p + 1)$ to q ;
$sons_p := sons_p + 1$

R$_p$: (* A basic message (M, g) is received *)
if $state_p = active$
then send **pas**$(g, 0)$ to Z
else $state_p, gen_p, sons_p := active, g, 0$

I$_p$: (* p becomes *passive* *)
{ $state_p = active$ }
send **pas**$(gen_p, sons_p)$ to Z ;
$state_p := passive$

RP: (* A **pas** message arrives at Z *)
receive **pas**(g, s) ; $ActCount[g] := ActCount[g] - 1$;
$ActCount[g + 1] := ActCount[g + 1] + s$;
if $\forall i : ActCount[i] = 0$ **then** send **term** to all processes

We present this algorithm as another illustration of our transformation, but Goldberg's remark: "it is not clear if there is any advantage to using the generational reference counting scheme instead of the weighted reference counting scheme" seems to apply equally to the resulting termination detection algorithms.

3.3 Mark–and–Sweep Garbage Collection

As explained in section 1.2.2, mark–and–sweep garbage collectors operate in consecutive cycles. In each cycle first all reachable objects are marked, and subsequently all unmarked objects are reclaimed. In this subsection it is shown how the garbage collection algorithm of Ben–Ari [Be84] can be transformed into a termination detection algorithm. (Actually, we use a variant of the algorithm described by Van de Snepscheut [Sn87]). This algorithm was designed to run concurrently with a single processor mutating the references contained in memory cells. Thus the copying of a reference is a single step where we have assumed so far that it consists of the sending and receipt of a message. When using Ben–Ari's collector, these two events must be assumed to be one single event, as it is in a distributed system with synchronous

communication. Therefore, in the remainder of this section we assume that message passing is *synchronous*. Let n be the number of processes.

3.3.1 Description of the Algorithm

In each cycle initially all nodes are white, and the following is done in a cycle:

1. Color all roots black.
2. Sequentially visit all nodes. For all black nodes, color the nodes to which it has references black.
3. Sequentially visit all nodes and count the number of black nodes.
4. If more black nodes were counted than in the previous round (more than the number of roots for the first round) go to step 2, otherwise to step 5.
5. Collect the white nodes and make all nodes white.

It is essential for the correctness of the algorithm that the basic program cooperates with the marker algorithm: whenever the basic program installs a new reference, it blackens the object to which it points. No cooperation is required when the basic program deletes a reference. The correctness proof of this garbage collection scheme is quite involved; two proofs are found in [Be84] and [Sn87].

3.3.2 Transformation into a Termination Detection Algorithm

In the scheme used to obtain a termination detection algorithm each object can have at most one reference, which is always a Z–reference. The transformation is straightforward. Rather than coloring the roots white at the end of each cycle and black again at the beginning of the next one, assume that the roots are always black by definition. The five steps of the algorithm are transformed as follows.

1. (Blacken the roots.) The roots (the processes A_p) are always black, so their color is not stored and this step is skipped.
2. (Blacken sons of black nodes.) In this step the virtual object Z need not be visited as it has no sons. The processes A_p are visited by arranging the processes in a (virtual) ring and passing a token along this ring. On this tour the token visits the processes in a "lazy" way: before actually visiting a process, it waits until the process is *passive*, and thus has no reference at all. This does not hinder the liveness of the termination detection, because there is no termination while a process is *active*. As a result of this strategy, no coloring is done in this round.

3. (Count black nodes.) There are $n+1$ processes, and the n roots are known to be black. It only needs to be determined whether any root has blackened Z, which is the case if a Z–reference has been installed (by the basic program) since the beginning of this cycle. To this end each process maintains a flag which is set when Z should be blackened according to the scheme (*viz.*, when becoming *active*). In order to see whether Z was blackened, the token again visits all processes, now testing whether any flag was set.

4. (Cycle completed?) If the second tour of the token reveals that no flag was set, Z is white and the number of black nodes is still n. In this case, go to step 5. If any flag was set (Z is black) there are now $n+1$ black nodes and the original algorithm would jump to step 2 in order to (try to) blacken more nodes. However in our case this is useless, as all nodes are black already, and we decide to also terminate the cycle. As Z is black, it cannot be collected, hence a new cycle of the collector must be started by resetting all the flags and returning to step 2.

5. (Collect.) The collection phase is entered with n black nodes, i.e., the virtual node Z is white, therefore termination can be signaled.

Essential for the termination detection algorithm is that a flag is set when a process is activated. (The meaning of this flag in the garbage collection scheme is "I blackened Z".) The termination detection algorithm repeatedly sends a token around the ring twice. In the first round the token only waits at each process until this process is *passive*. In the second round the token inspects and resets the flags. If no flag was set, termination is concluded, otherwise a new double tour of the token is initiated.

3.3.3 Correctness of the New Algorithm

In this section we formally present the new termination detection algorithm completely, and prove its correctness by means of an invariant. Assume the processes are numbered from 0 through $n-1$ and the processes have communication facilities so that processor q can send control messages to procesor $q-1 \pmod n$. The variable *tour* (values *first*, *second*) denotes whether the token is on the first or the second of the two tours. t denotes the current position of the token. Processes have a color (*black* or *white*), stored in $color_q$ for process q. The token has a color tc on its second tour.

The algorithm is initiated by process 0 by sending the token on its first tour to process $n-1$. A token visit during the first tour is described in action **T1**. It is enabled only when the process holding the token is *passive*, and consists of forwarding the token only (decrement t). If the token is at the end of the first tour it is whitened and sent on its second tour. A token visit during the second tour is described in action **T2**. The color of the visited node influences the color of the token, and is reset to white. When the token is not yet at the end of the

second tour it is forwarded. At the end of the tour, if the token is white termination is concluded, otherwise it is sent on its first tour again. Action $\mathbf{A}_p$ describes the (synchronous) activation of process q by process p and q's subsequent blackening. Action $\mathbf{I}_p$ describes how process p becomes *passive*.

T1: { $tour = first \wedge state_t = passive$ }
if $t > 0$
then $t := t - 1$
else $tour, t, tc := second, n - 1, white$

T2: { $tour = second$ }
if $color_t = black$ **then** $tc := black$; $color_t := white$;
if $t > 0$
then $t := t - 1$
else if $tc = white$
then *signal termination*
else (* Reinitialize procedure *)
$tour, t := first, n - 1$

$\mathbf{A}_p$: { $state_p = active$ }
$state_q := active$; $color_q := black$

$\mathbf{I}_p$: $state_p := passive$

During the first tour, the token visits a process only in *passive* state. Thus a process can be *active* "behind" the token only if it is reactivated after the visit, but this implies that the process is black. Black processes are reported in the second round, regardless of what the basic computation does. The principle of the algorithm is captured in the following predicate.

$$
\begin{aligned}
P \equiv \quad & tour = first \quad \wedge \quad (\forall q > t : state_q = passive \vee \exists q : color_q = black) \\
\vee \quad & tour = second \quad \wedge \quad (\forall q : state_q = passive \vee \\
& \qquad\qquad [tc = black \ \vee \ \exists q \leq t : color_q = black])
\end{aligned}
$$

Lemma 3.1 *P is an invariant of the algorithm.*

Proof. After initialization $tour = first$ and $t = n - 1$ so P holds. It is easily verified that each of the actions maintains P. □

Theorem 3.2 *The algorithm satisfies the safety and liveness criteria for termination detection algorithms.*

Proof. First suppose process 0 signals termination. This happens (see action **T2**) when the white token visits process 0 on its second tour, and $color_0 = white$. From P, all processes are *passive*.

Now suppose the termination condition holds. No more blackening of processes occurs, so the next complete second tour whitens all processes, and after the subsequent second tour termination is signaled. □

3.3.4 Discussion of the Algorithm

It is interesting to compare this algorithm with the similar algorithm by Dijkstra *et al.* [DFG83]. In that algorithm a process is blackened upon *sending* rather than upon *receiving*. Note that the subformula "$\forall q > t : state_q = passive$" of P is falsified when process $q > t$ is activated by some $p \leq t$. The consequence of blackening upon sending rather than receiving is, that a black process is certainly found *ahead of* the token in this case (while in the new algorithm the black process may be found behind the token). But then the two tours can be replaced by a single tour, and indeed the algorithm by Dijkstra *et al.* uses one tour only.

An alternative transformation of Ben–Ari's algorithm uses blackening upon sending. Indeed, sender and receiver cooperate atomically in the $\mathbf{A}_p$ action, which marks Z in Ben–Ari's algorithm, and this virtual marking can be flagged in the sender as well as in the receiver. With this modification the $\mathbf{A}_p$ action would be the following, and the reader may easily verify that this action also maintains invariant P above.

$\mathbf{A}_p$: { $state_p = active$ }
$state_q := active$; $color_p := black$

Unfortunately, invariant P is not strong enough to *exploit* the advantage of blackening upon sending, like in [DFG83]. We conclude that the transformation of Ben–Ari's garbage collection algorithm yields a termination detection algorithm which is very similar to Dijkstra's algorithm, but less efficient because it needs two control tours rather than one. It is possible, however, to optimize the scheme by combining the second tour of one cycle with the first tour of the next cycle. The combined action **T** describing the token visit is almost identical to **T2**:

```
T: { state_t = passive }
   if color_t = black then tc := black ; color_t := white ;
   if t > 0
     then t := t − 1
     else if tc = white
            then signal termination
            else (* Reinitialize procedure *)
                 tc, t := white, n − 1
```

4 Conclusions

In this paper we have presented a transformation of garbage collection schemes into termination detection protocols. Applying the transformation to known schemes, we

have derived several known termination detection algorithms, and three new ones: the Activity Counting algorithm, the Generational termination detection algorithm, and a "dual–tour" token algorithm for a ring of processors. Also the transformation is of a theoretical interest, and rises some further questions that will be addressed below.

4.1 Other Garbage Collection Algorithms

Virtually all garbage collection schemes can be transformed into sensible termination detection algorithms. Here we only sketch two more transformations, the reader is invited to complete the details and to apply the transformation to other garbage collection schemes (e.g., the well known algorithm by Dijkstra *et al.* [Dij78]).

A "classical" but now outdated garbage collection scheme consists in suspending the execution of the basic program when memory becomes short and run the garbage collector (of the mark–and–sweep type) while the program is stopped. Compared to on–the–fly garbage collection, synchronization and cooperation between the basic program and the collector is much simplified. The transformation of such a garbage collection algorithm yields a "freezing" termination detection algorithm where no reactivations are possible while the algorithm checks for the termination condition. In fact, one of the first published termination detection schemes was a freezing algorithm [Fr80].

In [St75] Steele describes a mark–and–sweep on–the–fly garbage collection algorithm. In this algorithm, when a reference from a marked to an unmarked object is installed, the marker process must visit the marked object again. In our transformation this principle means that when process p installs a reference to Z (i.e., becomes *active*), the termination detection algorithm must visit p again. This requirement can be realized in various ways. One way is to use a token visiting reactivated processes. Then the processes must keep specific information in order to record the identities of processes they reactivated (e.g., a vector with one component for each process).

4.2 A Different Transformation

The transformation of a garbage collection scheme into a termination detection algorithm as described in section 2 proved to be very useful—a number of interesting termination detection schemes resulted from its application. However, there exists a different transformation principle which is in some respects "dual" to the principle we used up to now. In this subsection we sketch that principle, the details, however, are omitted.

Each process p is transformed into a non–root object A_p. In addition, a single (virtual) root object R exists. It is assumed that initially only a single object A_0 is *active*, all other objects are initially *passive*. Throughout the computation the following equivalence must be guaranteed:

A_p is *active* $\Leftrightarrow$ R has an A_p–reference.

This equivalence can easily be realized by a reference counting algorithm: The local reference counter RC_p is incremented when process A_p is activated, and decremented when it becomes *passive*. Thus, R–references are only virtual references, they need not be implemented.

In order to detect termination, we want to maintain the following property:

A_0 is garbage $\Rightarrow$ the termination condition holds.

Using this property a (distributed) garbage collection algorithm can detect the termination condition when collecting A_0. The property is maintained by the following two rules:

1. Each basic message from object A_p to object A_q contains an A_p–reference.
2. When object A_q receives a message containing an A_p–reference it inserts that reference if it has no reference to any object, otherwise it immediately deletes the A_p–reference. If A_q is activated, RC_q is incremented.

The reader may easily verify that no cycles are formed and that if an object A_p is *active* then there exists a reference path from A_p (and consequently also from R) to A_0. In order to guarantee the liveness property, the following rules should be observed in addition:

3. When object A_p becomes *passive*, RC_p is decremented. (That is, the virtual A_p–reference in R is deleted.)
4. When the reference counter RC_p of object A_p drops to zero and A_p has an A_q–reference, that reference is deleted ("recursive freeing" as part of the virtual collection of the garbage object A_p).

When Lermen and Maurer's garbage collection algorithm is used (see section 3.1), the two messages **ack** and **cop** can be merged into a single message. This is the case because A_p sends a reference pointing to itself. The only effect of the subsequent execution (in arbitrary order) of the receive actions $\mathbf{RA}_p$ and $\mathbf{RC}_p$ is to increment $aR_p(o)$ which is used as a guard for action $\mathbf{DR}_p$. Since A_p (virtually) sends **inc** to itself, action $\mathbf{RI}_p$ is executed locally in A_p when sending a basic message.

The algorithm resulting after removal of the manipulation of references is simple. Whenever an object becomes *active* or sends a basic message it increments a local counter. The counter is decremented when the object becomes *passive* or when a decrement control message is received. Decrement messages are either sent by rule 2 or by rule 4. The FIFO property is not required because the channel for which it was necessary in the original garbage collection algorithm no longer exists. Termination is detected when the counter of A_0 drops to 0.

The resulting algorithm is already known; it is Dijkstra and Scholten's termination detection scheme for diffusing computations [DS80].

4.3 Reversed Transformation

In this section it is indicated how a termination detection algorithm can be transformed into a reference counting garbage collection scheme. The aim of a reference counting algorithm is to collect an object o when all o–references (in objects) have been deleted and no more o–references are in transit (in copy messages). A similarity to the termination detection problem is observed when activity of objects is defined suitably.

An object is defined to be *o–active* if it holds an o–reference and *o–passive* otherwise, and a message is called an o–activation message if it carries an o–reference. Under these definitions, an *o–passive* object becomes *o–active* only upon receipt of an o–activation message, and only *o–active* objects send o–activation messages. The behavior of the computation is according to the skeleton in section 1.1.1, so that the *o–termination condition*, defined as

no process is *o–active* and no o–activation messages are in transit

is stable and can be detected by a termination detection algorithm. Furthermore

there are no o–references $\Leftrightarrow$ the o–termination condition holds. (RT)

To arrive at a reference counting algorithm, a termination detection algorithm is superimposed on the o–reference handling. When the o–termination condition is established, o is collected. For each object a separate instance of the termination detection algorithm is executed concurrently.

Not all termination detection algorithms can be reasonably used in this construction. Several considerations must be taken into account.

Centralized versus Distributed Control. It is not a drawback if an algorithm is chosen in which one process plays a special role, such as initiating the algorithm. The object o itself is a natural candidate to play this role. The central object could however become a bottleneck if its intervention is needed in every basic communication (as in the Activity Counting or Generational termination detection algorithm).

No Probe–Based Algorithms. The set of objects can be very large and varies due to creation and collection of objects. Therefore it is not feasible to use a termination detection algorithm in which *all* processes in the system take part. Rather, the activity of the algorithm should be restricted to processes that take part in the basic computation also. The algorithm of [DS80], the Credit, Activity Counting, and Generational termination detection algorithm all have this property, but probe–based algorithms are ruled out.

Early Termination. A process p holding or having held an o–reference may detect that it is garbage itself and must be collected. Therefore the termination detection algorithm must allow processes to terminate locally even while the computation as a whole has not yet terminated. The algorithm of [DS80] does not have this property: an "engaged" process must remain in the system as long as any of its descendants remain *active*.

These considerations differ from those that are usually taken into account when a termination detection algorithm is designed. For example, it is usually preferred that processes participate in the termination detection procedure only while they are *passive*, but this property probably conflicts with the possibility of early termination. The termination detection algorithms that we have derived from reference counting schemes present themselves as candidate algorithms, but their transformation yields no new algorithms, and all suffer from the "bottleneck" disadvantage. Current research addresses the design of a new termination detection algorithm, based on the algorithm in [DS80] but with early termination, and its transformation into a reference termination scheme.

This transformation is different from the transformation considered in [TTL88], where it was observed that detecting the termination of the marking phase emerges as a natural subproblem in mark–and–sweep garbage collectors. It was shown that the choice of a particular termination detection algorithm has a major influence on the resulting garbage collection algorithm. Although virtually all termination detection algorithms can be used for that purpose, only a subclass of the garbage collection algorithms is obtained.

4.4 Related Problems

The termination detection problem is an instance of a class of detection problems in distributed systems. Communication deadlock detection is a generalization where also a part of the network can be terminated [Na86]; distributed infimum approximation is a generalization where the "property" to be detected takes values from any partially ordered domain, rather than just *passive* or *active* [Te86, Ma90].

Deadlock Detection. In the communication deadlock problem, for each *passive* process a subset of the processes is determined at the moment it becomes *passive*. The process can become *active* only by receiving a message from a process in this subset. The termination detection problem is obtained, when each process always chooses the full set of processes. Thus, both the garbage collection problem and the communication deadlock detection problem seem to "dominate" the termination detection problem. This rises the question whether our approach can be generalized to detection of communication deadlocks.

Distributed Infimum Approximation. In the distributed infimum approximation problem an arbitrary partially ordered domain X with the *infimum* operator $\wedge$ replaces the two–valued domain $\{active, passive\}$. The "state" x_p of process p is a value from X, and the messages of the basic computation are tagged with values from X. The handling of message tags and states in the operations of the basic computation satisfies the following rules.

$\mathbf{S}_p$: (* Send basic message *)
send a message (M, x_p)

$\mathbf{R}_p$: (* A basic message arrives *)
receive message (M, x) ; $x_p := x_p \wedge x$

$\mathbf{I}_p$: (* Internal increase of x *)
$\{\ x_p < x\ \}$ $x_p := x$

The problem is to approximate the global state function F, defined as the infimum of all states and tags of messages. It follows from these actions that this function is monotonically increasing.

It would be interesting if our current construction could be generalized to obtain Distributed Infimum Approximation algorithms. To this end, instead of the single object Z a directed graph G_X of objects could be defined, reflecting the structure of X. The actions of the basic computation must then be formulated as reference manipulation, such that the growth of F is reflected by objects of G_X becoming garbage.

4.5 Legal Aspects

We have shown that termination detection algorithms are obtained as suitable instantiations of garbage collection schemes. Supplying a particular scheme, our transformation yields a particular termination detection algorithm. It may happen that the resulting algorithm was found independently already. This is the case with the weighted reference counting scheme, which yields the Credit Recovery algorithm for termination detection, see section 3.2.

Commercial use of the weighted reference counting scheme by Watson and Watson is protected by a patent [EPO]. Does the patent now cover the Credit Recovery algorithm? The patent describes the invention in "a computer system having storage means containing memory cells, at least some of which contain pointers to others", so that it seems to us that it does not cover applications of the invention not using pointers explicitly. Ben Lian argued however, that "since the algorithm itself is patented, then any technique which makes use of weighted reference counts in any form is covered by the patent. This is regardless of different terminology and/or strategy used to split reference weights."

In recent years more papers described general transformations of solutions to one problem into solutions to another problem. The implication a patent on such a solution can have in general is too complicated for us and we are glad to leave it as food for lawyers.

References

[Be84] Ben–Ari, M., *Algorithms for On–the–fly Garbage Collection*, ACM Trans. on Prog. Lang. and Systems 6 (1984) 333–344.

[Be89] Bevan, D.I., *An Efficient Reference Counting Solution to the Distributed Garbage Collection Problem,* Parallel Computing 9 (1989) 179–192.

[CL85] Chandy, K.M., L. Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. on Computer Systems 3 (1985) 45–56.

[CM86] Chandy, K.M., J. Misra, *How Processes Learn*, Distributed Computing 1 (1986) 40–52.

[CMH83] Chandy, K.M., J. Misra, L.M. Haas, *Distributed Deadlock Detection*, ACM Trans. on Computer Systems 1 (1983) 144–156.

[Co60] Collins, G.E., *A Method for Overlapping and Erasure of Lists*, Comm. ACM 3 (1960) 655–657.

[CV90] Chandrasekaran, S., S. Venkatesan, *A Message–Optimal Algorithm for Distributed Termination Detection*, Journal of Parallel and Distributed Computation 8 (1990) 245–252.

[DFG83] Dijkstra, E.W., W.H.J. Feijen, A.J.M. van Gasteren, *Derivation of a Termination Detection Algorithm for Distributed Computations,* Inf. Proc. Lett. 16 (1983) 217–219.

[Dij78] Dijkstra, E.W., L. Lamport, A.J. Martin, C.S. Scholten, E.F.M. Steffens, *On–the–fly Garbage Collection: An Excercise in Cooperation*, Comm. ACM 21 (1978) 966–975.

[DS80] Dijkstra, E.W., C.S. Scholten, *Termination Detection for Diffusing Computations*, Inf. Proc. Lett. 4 (1980) 1–4.

[EPO] European Patent Office, *Garbage Collection in a Computer System*, European Patent Application no 86309082.5.

[Fr80] Francez, N., *Distributed Termination*, ACM Trans. on Prog. Lang. and Systems 2 (1980) 42–55.

[Go89] Goldberg, B., *Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme*, ACM SIGPLAN Notices 24 (July 1989) 313–321.

[LM86] Lermen, C.–W., D. Maurer, *A Protocol for Distributed Reference Counting*, ACM Conference on Lisp and Functional Programming, Cambridge, 1986, pp. 343–354.

[Ma87] Mattern, F., *Algorithms for Distributed Termination Detection*, Distributed Computing 2 (1987) 161–175.

[Ma89] Mattern, F., *Global Quiescence Detection Based on Credit Distribution and Recovery,* Inf. Proc. Lett. 30 (1989) 195–200.

[Ma90] Mattern, F., *Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non–FIFO Systems*, Tech. Rep. SFB124–24/90, Kaiserslautern University, 1990.

[McC60] McCarthy, J., *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Comm. ACM 3 (1960) 184–195.

[Mi83] Misra, J., *Detecting Termination of Distributed Computations Using Markers*, Proc. of the 2nd ACM Symp. on Principles of Distributed Computing, Montreal, Quebec, 1983, pp. 290–294.

[Na86] Natarajan, N., *A Distributed Scheme for Detecting Communication Deadlocks*, IEEE Trans. on Software Engineering SE–12 (1986) 531–537.

[Ru88] Rudalics, M., *Multiprocessor List Memory Management*, Technical Report RICS–88–87.0, Research Institute for Symbolic Computation, J. Keppler University, Linz, 1988.

[SF86] Shavit, N., N. Francez, *A New Approach to Detection of Locally Indicative Stability*, in: L. Kott (ed.), *Proceedings ICALP 1986*, Lecture Notes in Computer Science 226, Springer–Verlag, 1986.

[Sn87] Van de Snepscheut, J.L.A, *"Algorithms for On–the–fly Garbage Collection" Revisited*, Inf. Proc. Lett. 24 (1987) 211–216.

[St75] Steele, G.L., *Multiprocessing Compactifying Garbage Collection*, Comm. ACM 18 (1975) 495–508.

[ST88] Schoone, A.A., G. Tel, *Transformation of a Termination Detection Algorithm and its Assertional Correctness Proof*, Technical Report RUU–CS–88–40, Dept. of Computer Science, Utrecht University, 1988.

[Te86] Tel, G., *Distributed Infimum Approximation*, Technical Report RUU–CS–86–12, Dept. of Computer Science, Utrecht University, 1986.

[Te90] Tel, G., *Total Algorithms*, Technical Report RUU–CS–88–16, Dept. of Computer Science, Utrecht University, 1988. Also in: Algorithms Review 1 (1990) 13–42.

[TTL88] Tel, G., R.B. Tan, J. van Leeuwen, *The Derivation of Graph Marking Algorithms from Distributed Termination Detection Protocols*, Science of Computer Programming 10 (1988) 107–137.

[We79] Weng, K.–S., *An Abstract Implementation for a Generalized Dataflow Language*, Technical Report MIT/LCS/TR–228, Massachusetts Institute of Technology, 1979.

[WW87] Watson, P., I. Watson, *An Efficient Garbage Collection Scheme for Parallel Computer Architectures*, in: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (eds.), Proceedings Parallel Architectures and Languages Europe, vol. II, Lecture Notes in Computer Science 259, Springer–Verlag, 1987, pp. 432–443.