Transformation of a Termination Detection Algorithm and its Assertional Correctness Proof

Anneke A. Schoone and Gerard Tel

RUU–CS–88–40
December 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Transformation of a Termination Detection Algorithm and its Assertional Correctness Proof

Anneke A. Schoone and Gerard Tel

Technical Report RUU–CS–88–40
December 1988

Department of Computer Science
University of Utrecht
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands.

TRANSFORMATION OF A TERMINATION DETECTION ALGORITHM AND ITS ASSERTIONAL CORRECTNESS PROOF

Anneke A. Schoone and Gerard Tel

Department of Computer Science, University of Utrecht,
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract. A stepwise transformation of a termination detection algorithm to a distributed infimum approximation algorithm is presented. The algorithm is transformed together with its invariants and correctness proof. Thus a (fairly complex) algorithm for the distributed infimum approximation problem is obtained together with a formal (assertional) proof of correctness, whereas previous algorithms for distributed infimum approximation were proved correct by operational reasoning.

1. Introduction. Assertional correctness proofs have been recognized as a reliable means for the verification of distributed programs. These correctness proofs are based on the proven invariance of propositions about the system's state. Operational correctness arguments, which are based on reasoning about executions of the programs, have all too often turned out to be error prone. Recently a quite complicated algorithm for the termination detection problem due to Mattern [Ma87] was verified by the present authors using assertional methods [ST88].

The problem of *distributed infimum approximation* [Te86] was formulated as an abstraction of several known problems in the field of distributed computing, including termination detection. Other problems include global virtual time approximation [Je85] and an update problem in Hughes' distributed garbage collecting algorithm [Hu85]. As termination detection is a special case of distributed infimum approximation, termination detection algorithms can be obtained by instantiating general distributed infimum approximation algorithms. On the other hand, the distributed infimum approximation algorithms in [Te86] were inspired by existing termination detection algorithms. In [Te89] it was shown that in order to verify a general distributed infimum approximation algorithm, it is sufficient to verify a derived termination detection algorithm.

This paper further develops the connection between termination detection and distributed infimum approximation. A termination detection algorithm is transformed into a distributed infimum approximation algorithm, together with its assertional correctness proof. Although we describe the transformation of one particular algorithm rather than a general transformation that can be applied to any algorithm, we believe that other termination detection algorithms are candidates for a similar transformation also. As a result of the transformation we obtain a distributed infimum approximation algorithm together with an assertional correctness proof. The

---

The work of the second author was supported by the Foundation for Computer Science (SION) of the Netherlands Organization for Scientific Research (NWO). The email addresses of the authors are anneke@ruuinf.uucp and gerard@ruuinf.uucp.

algorithms for distributed infimum approximation in [Te86] were verified using operational reasoning. Operational reasoning attempts to check all possible executions to verify that they satisfy the requirements. It is felt to be error prone.

This report is organized as follows. Preliminaries and problem definitions are given in section 2. In section 3 we present the termination detection algorithm to which the transformation is applied. In section 4 this algorithm is transformed step-wise into a distributed infimum approximation algorithm. Section 5 contains the conclusions.

2. Preliminaries. We give some mathematical background concerning posets and infimums in section 2.1. In section 2.2 we discuss models of computation. In sections 2.3 and 2.4 we present the termination detection and the distributed infimum approximation problem, respectively. In section 2.5 we discuss the algorithm we will derive.

2.1. Infimums. A binary relation $\leq$ on a set A is a partial ordering if it is transitive, antisymmetric, and reflexive. The resulting structure $(A, \leq)$ is called a partially ordered set or poset. If the poset $(A, \leq)$ contains a (unique) element u such that for all $a \in A$, $u \leq a$, then this element is called "bottom", denoted as: $\perp$. Not all posets contain a bottom element, but we can always add a bottom element to a poset (for example, $-\infty$ can be added as a bottom element to the set of all integers with the usual ordering).

In a poset, an element c is called the infimum of two elements a and b if it is the (unique) element satisfying:

(1) $c \leq a$ and $c \leq b$,

(2) for all z with $z \leq a$ and $z \leq b$, $z \leq c$.

We denote this by $c = \inf(a,b)$. The infimum of two elements does not necessarily always exist, but we restrict ourselves to posets in which the infimum exists for any two elements.

It can be shown that the binary operator "inf" is commutative, associative, and idempotent. Thus the notation $\inf B$, the infimum of B, for finite subsets B of A, is unambiguous and will be used as a shorthand notation for the repeated infimum of all elements of B.

The importance of the abstract notion of infimum lies in the fact that for any commutative, associative, and idempotent binary operator $\Box$ on a set A, there is a partial ordering on A such that for all x and y in A, $x \Box y = \inf(x,y)$ [Kl43]. It turns out that many network dependent values can be viewed as the infimum of certain local values of the processors in the network. For example termination of a computation or the election problem can be viewed as such.

2.2. Models of computation. Throughout this paper we do not specify complete algorithms, but give so-called program skeletons. These are generic descriptions for classes of algorithms all of which have some common underlying structure. A program skeleton consists of a number of operations, each of which is given by a piece of program. An operation is viewed as an atomic action, i.e., it is not interruptable. We do not specify anything about an assumed order in which the operations must take place, but operations can contain a so-called guard: a boolean expression between braces { }. An operation can only be executed if its guard is true. For example, a process can only execute the code for receiving a message if there is a message present to receive.

From a program skeleton with a set of atomic operations, various tailor-made algorithms can be built. For example, in the sequel we will have atomic operations which have a guard that tests for the presence of a token in the process wanting to execute the code. This can be used to build so-called token-based algorithms in which a token is circulated which collects necessary information. As nothing is specified about how token-visits are organized, we could choose for some predestined tour for the token (e.g. a ring) in one algorithm, and we could let the token take a tour which depends on the information collected so far in another. We could even choose to implement a token-visit by a message sent to some central process, which collects and computes the information for the whole network.

In the proofs we use the method of assertional verification or system-wide invariants, first introduced by Krogdahl [Kr75] and Knuth [Kn81]. The idea is that if a relation (between process variables for example) holds initially, and is kept invariant by all possible operations, then it will always hold in the distributed system, in whatever order the operations take place in an actual execution. The proof technique is rather different from "operational reasoning" which is built on checking all possible executions. The problem with operational reasoning for correctness proofs is that it is almost impossible not to overlook some odd coincidence of events which might render a proof invalid.

We distinguish different models of computation, that is different ways to group actions as atomic operations. In the message-driven model all operations are triggered by the receipt of a message. Thus in general the program skeleton consists of only one operation B and has the following form. (We give the name of the process p where the operation takes place as a subscript.)

B_p: {a message arrives at p }
begin receive the message; compute; send all appropriate messages **end**

In the so-called transactional model, the sending of messages and internal computations can be done spontaneously, thus allowing more freedom in an actual algorithm on the one hand, and making correctness proofs more intricate on the other hand, simply because of this extra freedom. A program skeleton in the transactional model looks like this.

BS_p: **begin** send a message **end**

BR_p: {a message arrives at p }
begin receive the message; compute **end**

BI_p: **begin** compute **end**

Note that the basic operation in the message-driven model can be viewed as a special combination of the send, receive and internal operations in the transactional model. Thus the message-driven model is just a restriction of the transactional model.

2.3. Termination detection. In the problem of termination detection we assume that there is some underlying basic computation, carried out by a finite set of processes in a distributed manner, and that the processes somehow should detect that the computation has terminated. The problem arises when messages that are in transit in the distributed system are unobservable, but can influence the computation, when they will be received. The termination detection algorithm which was proved correct in [ST88] was given for the message-driven model of

computation. In this model the basic computation consists of receiving a message, doing some computations, deciding what messages should be sent to which processes, and sending those messages. If we denote the set of processes to which a message is to be sent as B, a process in this set as b, and the message to be sent as $<m_b>$, then the only operation of the basic computation B_p has the form:

B_p: {a message $<m>$ arrives at p}
begin receive $<m>$; compute; **forall** $b \in B$ **do** send $<m_b>$ to b **od end**

Termination of a message-driven algorithm is characterized by

$TERM_{md}$ = there are no messages underway.

Once the situation of $TERM_{md}$ is reached, the basic operations B for all processes are disabled, as no messages can arrive at a process. Thus this situation will endure forever, and $TERM_{md}$ is a stable property of the distributed system.

In the more general transactional model of computation it is assumed that internal computation and the sending of messages take place separately from receiving messages. If termination of a computation is to be detected, it is assumed that a process is either in *passive* or in *active* state, with respect to this computation. Only *active* processes can send messages, and only the receipt of a message can make a process *active*. The state of a process p is maintained in the variable x_p. Formally the send, receive and internal actions are described as follows.

BS_p: $\{x_p = active\}$
begin send $<m>$ to some process b **end**

BR_p: {a message $<m>$ arrives at p}
begin receive $<m>$; $x_p := active$ **end**

BI_p: **begin** $x_p := passive$ **end**

Termination of the computation in this model is characterized by:

$TERM_{tr0}$ = there are no messages underway and $\forall p\ x_p = passive$.

Again it is easily seen that $TERM_{tr0}$ is a stable predicate: when it holds BS actions are disabled, hence do not occur, BR actions are also disabled, and BI actions have no influence on the system state whatsoever. It makes no difference to assume that, besides activating messages, a process can also send messages while *passive*; as long as these messages are distinguishable from activating messages and do not cause their receiver to become *active*. That no activating messages are underway is equivalent to all messages underway having *passive* status. The actions are now described as follows.

BS_p: **begin** send $<m, x_p>$ to some process b **end**

BR_p: {a message $<m, x>$ arrives at p}
begin receive $<m, x>$; **if** $x = active$ **then** $x_p := active$ **fi end**

BI_p: **begin** $x_p := passive$ **end**

In this formulation, termination is characterized by:

$TERM_{tr}$ = all messages underway are *passive* and $\forall p\ x_p = passive$.

Clearly the two models are equivalent with regard to termination. The problem of termination detection is as follows. An algorithm should be superimposed upon the basic computation, which allows the processes to detect that termination (denoted as TERM if no model of computation is specified) occurs. This detection can be formalized by sending a special message, entering a special state, etc. For convenience we assume that a process detects termination by executing a special statement *detect*. The correctness requirements for the superimposed algorithm are the following.

(Independence)	It should not influence the execution of the basic computation.
(Safety)	If any process executes *detect*, then TERM holds.
(Liveness)	If TERM holds, then within finite time some process executes *detect*.

Note that it is possible to fulfill these requirements only because TERM is a stable predicate.

2.4. Distributed infimum approximation. We consider the distributed infimum approximation problem, as introduced in [Te86], for the most general, i.e., the transactional model. Processes are assumed to maintain a variable x, which is an element of a partially ordered set A. Its current value is appended to any message a process sends. Upon receiving a message, the receiver's value is replaced by the infimum of the old and the received value. Internal actions, insofar they influence the value, may only increase the value. Formally, the effect of the basic computation on the value is described as follows.

BS_p: **begin** send $<m, x_p>$ to q **end**

BR_p: {a message $<m, x>$ arrives at p}
begin receive $<m, x>$; $x_p := \inf(x_p, x)$ **end**

BI_p: **begin** choose $x \geq x_p$; $x_p := x$ **end**

The distributed infimum that we seek to "approximate" is defined by:

$$F = \inf(\{x \mid <m, x> \text{ is in transit}\} \cup \bigcup_p \{x_p\}).$$

The rules for maintaining values and sending messages ensure that F is a monotone function. Because of this monotonicity it makes sense to formulate the approximation problem for F. This approximation can be formalized by sending a special message, setting a dedicated variable to the value of the approximation, etc. For convenience we assume that a process yields a value k as an approximation to F by executing a statement *yield* (k). The correctness requirements for the superimposed algorithm are:

(Independence)	It should not influence the execution of the basic computation.
(Safety)	If any process executes *yield* (k), then $F \geq k$.
(Liveness)	If $F \geq k$, then within finite time some process executes *yield* (k'), for some $k' \geq k$.

Finally, we note that indeed the termination detection problem can be seen as a special case of the distributed infimum approximation problem. Let $A = \{active, passive\}$, ordered by defining $active \leq passive$. Under this instantiation the actions described in this section read as those in section 2.3. The condition $TERM_{tr}$ of section 2.3 now reads "$passive \leq F$" and from this it can be seen that if we read *detect* as *yield* (*passive*), the correctness criteria for distributed infimum approximation and termination detection algorithms match.

2.5. The new algorithm. Before actually deriving the new distributed infimum approximation algorithm we present a short discussion of the algorithm and its merits. In contrast to earlier algorithms, which were verified using operational reasoning, the new algorithm is verified using assertional reasoning. A major advantage of the new algorithm is the potentially lower message complexity. Token exchanges can be restricted to processes of interest, namely, processes that participate in the basic computation. A major disadvantage of the algorithm is the high volume of information that is maintained both in the token and in processes.

The algorithm is based on a (virtually) circulating token rather than repeated observation of all processes. Earlier distributed infimum approximation algorithms [Te86] were inspired by existing termination detection algorithms and inspected all processes again before yielding a new approximation, even if the computation was concentrated in only a few of them. In [ST88] a termination detection algorithm was analyzed which was based on a circulating token. Safety of the algorithm was shown to be independent of the tour this token makes through the network. A mechanism for determining a tour was presented, such that the token only visits processes that actually take part in the computation.

Attempts to apply the same principle to distributed infimum approximation led to the new algorithm. It was shown in [Te89] that, once given the new algorithm, the verification of its safety property can be reduced to the verification of the safety of a termination detection algorithm. In this paper we show that the algorithm itself can be obtained by transforming the termination detection algorithm. Again we concentrate on the safety properties of the algorithm. The design of a mechanism to determine the tour of the token is a very intricate matter in the case of a general distributed infimum approximation algorithm. This question will be addressed in section 5.1.

The volume of information is high and the algorithm complex because the token contains a complete snapshot of the basic computation. It consists of x-values of all processes and sets of messages that are in transit. It should be noted that the information in the token is "old", that is, each piece of it is related to an earlier system state. In fact, the information does not even constitute a "consistent snapshot" in the sense of [LY87]. We shall demonstrate however, that the information in the token allows for a safe approximation of the value of F. In order to maintain the necessary information between token-visits, a process p stores sent and received messages in a local bag variable L_p. L_p contains separate subbags $L_p[q]$ for every addressee q. In a token-visit the contents of L_p and x_p are copied to the token. Formally, the algorithm (including the basic computation) is completely described by the following program skeleton. (For the precise meaning of several symbols see section 4.)

Initially
$\forall p$: $X[p] \leq x_p$, $Q[p] = \{x \mid <m,x>$ initially underway to $p\}$,
$\forall q$: $L_p[q] = \emptyset$.

BS_p: **begin** send $<m, x_p>$ to q; $L_p[q] := L_p[q] \cup \{x_p\}$ **end**

BR_p: {a message $<m,x>$ arrives at p}
begin receive $<m,x>$; $L_p[p] := L_p[p] \cup\!\!- \{x\}$; $x_p := \inf(x, x_p)$ **end**

BI_p: **begin** choose $x \geq x_p$; $x_p := x$ **end**

T_p: {process p holds the token (Q,X)}
begin forall q **do** $Q[q] := Q[q] \cup L_p[q]$; $L_p[q] := \varnothing$ **od**;
$X[p] := x_p$
end

D_p: {process p holds the token (Q,X)}
begin *yield* $(\inf(\inf X, \inf_p \inf Q[p]))$ **end**

3. The termination detection algorithm. In this section we present a termination detection algorithm for transactional computations, together with its assertional correctness proof. The algorithm is a generalization of the transactional model of the algorithm in [ST88], which was suited for message-driven computations only. The algorithm from [ST88] and its transactional generalization are given in section 3.1. The correctness proof follows in section 3.2.

3.1. Presentation of the algorithm. In the program skeleton of [ST88] each process p counts all messages it receives, and it counts how many messages it sent to all possible destinations in an array L_p. L_p has an entry for every destination, and messages received by process p are counted negatively in $L_p[p]$. There is a token Q, which collects all these counts. Q also has an entry for every destination. If a process p has the token Q, it adds L_p to Q and sets L_p to $\vec{0}$. The idea is that if there are as many messages received as sent, there can be no more messages underway. The claim which is proved in [ST88] is that if all entries in Q are ≤ 0, there is termination. Thus a process which holds the token Q could "detect" this state. We recall the program skeleton in the message-driven model ([ST88]):

Initially
$\forall p$: $Q[p]$ = the number of messages initially underway to p,
$\forall q$: $L_p[q] = 0$.

B_p: {a message $<m>$ arrives at p}
begin receive $<m>$; $L_p[p] := L_p[p] - 1$; compute;
forall $b \in B$ **do** send $<m_b>$ to b; $L_p[b] := L_p[b] + 1$ **od**
end

T_p: {process p holds the token (Q)}
begin $Q := Q + L_p$; $L_p := \vec{0}$ **end**

D_p: {process p holds the token (Q)}
begin if $Q \leq \vec{0}$ **then** *detect* **fi end**

$TERM_{md}$ = there are no messages underway.

In the next section we allow that *passive* processes send messages, albeit *passive* ones, but that only *active* messages can activate other processes. Hence in counting messages, we now distinguish between *active* and *passive* messages. As *passive* messages do not activate other processes, it suffices to count *active* messages. However, we need to extend the token with an array X with a value for each process p, to record whether a process visited was still *active* or not, as *active* processes now can spontaneously send *active* messages. The resulting program skeleton then is:

Initially
$\forall p$: $X[p] = active \vee X[p] = x_p$,
$Q[p]$ = the number of *active* messages initially underway to p,
$\forall q$: $L_p[q] = 0$.

BS_p: **begin** send $<m, x_p>$ to q;
if $x_p = active$ **then** $L_p[q] := L_p[q]+1$ **fi**
end

BR_p: {a message $<m, x>$ arrives at p}
begin receive $<m, x>$;
if $x = active$ **then** $L_p[p] := L_p[p]-1$; $x_p := active$ **fi**
end

BI_p: **begin** $x_p := passive$ **end**

T_p: {process p holds the token (Q, X)}
begin $Q := Q + L_p$; $L_p := \vec{0}$; $X[p] := x_p$ **end**

D_p: {process p holds the token (Q, X)}
begin if $Q \leq \vec{0}$ and $X = \vec{passive}$ **then** *detect* **fi end**

$TERM_{tr}$ = all messages underway are *passive* and $\forall p$ $x_p = passive$.

Next we extend this program skeleton with the auxiliary and ghost variables necessary for the proof with system-wide invariants, as was done in [ST88]. The variables R, S, Q, f_p, and Ω are as in the original extended program skeleton and are explained below. We now count separately all messages received, and split out the count according to the origin of the message. For ease of presentation, we notate all counts of received messages in one two-dimensional array R. Thus $R[p, q]$ contains the number of messages q received from p since the last token visit to q. Hence q only has access to its own column of R. Likewise, all messages sent are counted in the two-dimensional array S. $S[p, q]$ contains the number of messages p sent to q since the last token visit to p. Hence p has only access to its own row of S. Likewise, Q also now is two-dimensional. $Q[p, q]$ contains the number of messages sent from p to q minus the number of messages q received from p, as far as accumulated by the token. Thus Q can contain negative entries.

In f_p a process p records where the first *active* message received since the last token visit came from, thus recording which process activated it. The variable Ω is a queue which contains for each process p two elements, namely a label b_p and a label t_p. The order in the queue Ω represents an ordering in important events which have taken place. A label t_p represents the last token visit to process p (i.e. an execution of operation T_p), and a label b_p represents the last activation of p, i.e. a first execution of operation BR_p for an *active* message after a token visit. The end of the queue Ω represents the largest elements or the latest events. We will write $t_p < b_q$ for: the last token visit to p occurred before the last activation of q.

M is a ghost variable containing in an entry $M[p, q]$ the number of *active* messages underway from process p to process q, while the ghost variable N contains in entry $N[p]$ the value of x_p. These variables are only introduced to make stating invariants simpler, we will

state the changes in these variables in comment. Note that everywhere we only count *active* messages.

Initially
$\forall p$: $X[p] = x_p \vee X[p] = active$, $f_p = nil$,
$\forall q$: $S[p,q] = 0$, $R[p,q] = 0$, $Q[p,q] = M[p,q] \geq 0$, and $b_p < t_q$ in Ω.

BS_p: **begin** send $<m, x_p>$ to q;
if $x_p = active$
then $S[p,q] := S[p,q]+1$ (* $M[p,q] := M[p,q]+1$ *)
fi
end

BR_p: {a message $<m,x>$ from q arrives at p}
begin receive $<m,x>$;
if $x = active$
then $R[q,p] := R[q,p]+1$; (* $M[q,p] := M[q,p]- 1$ *)
$x_p := active$; (* $N[p] := active$ *)
if $f_p = nil$ **then** $f_p := q$; move b_p to the end of Ω **fi**
fi
end

BI_p: **begin** $x_p := passive$ **end** (* $N[p] := passive$ *)

T_p: {process p holds the token (Q,X)}
begin $f_p := nil$; move t_p to the end of Ω; $X[p] := x_p$;
forall q
do $Q[p,q] := Q[p,q]+S[p,q]$; $S[p,q] := 0$;
$Q[q,p] := Q[q,p]+R[q,p]$; $R[q,p] := 0$
od
end

D_p: {process p holds the token (Q,X)}
begin if $\forall r,s\ Q[r,s] \leq 0 \wedge \forall r\ X[r] = passive$ **then** *detect* **fi end**

$TERM_{tr}$ = all messages underway are *passive* and $\forall p\ x_p = passive$,
or, equivalently,
$TERM_{tr}$ = $\forall p,q\ M[p,q] = 0 \wedge \forall p\ N[p] = passive$.

Note that all message counters only count *active* messages, and that the state of messages cannot be changed. In other words, *passive* messages have no effect whatsoever.

3.2. Correctness proof for transactional termination detection. Since in the transactional model the active state is not hidden, as in the message-driven model, we will have to extend the invariants of [ST88] to reflect this.

Lemma 3.1. For all p and q the following holds invariantly.
(1) $R[p,q] \geq 0$,

(2) $S[p,q] \geq 0$,
(3) $Q[p,q]+S[p,q] = M[p,q]+R[p,q]$,
(4) $N[p] = passive \vee X[p] = active \vee f_p \neq nil$.

Proof. (1). Initially $R[p,q] = 0$. Only operations BR_q and T_q can change $R[p,q]$. The first does not decrease $R[p,q]$ and the second sets $R[p,q] = 0$ again. Hence (1) holds.
(2). Initially $S[p,q] = 0$. Only BS_p and T_p can change $S[p,q]$, the first does not decrease $S[p,q]$ and the second sets $S[p,q] = 0$ again. Hence (2) holds.
(3). Initially $Q[p,q] = M[p,q]$ and as $R[p,q] = S[p,q] = 0$, the relation holds. If BS_p increases $S[p,q]$, $M[p,q]$ is increased by the same amount. If BR_q increases $R[p,q]$, $M[p,q]$ is decreased by the same amount. BI actions do not change any message counters. T_p adds $S[p,q]$ to $Q[p,q]$ and sets $S[p,q] = 0$, thus keeping the sum $Q[p,q]+S[p,q]$ the same. T_q subtracts $R[p,q]$ from $Q[p,q]$ and sets $R[p,q] = 0$, thus keeping the difference $Q[p,q]-R[p,q]$ the same. Hence (3) holds.
(4). Initially $X[p] = N[p]$ or $X[p] = active$, hence the relation holds. BS_p and D_p do not change any of the variables involved. If BR_p makes $N[p] = active$, it sets $f_p \neq nil$. BI_p can set $N[p]$ to *passive*, which is all right. T_p sets $f_p = nil$ and $X[p]$ to $N[p]$. Since they are either *active* or *passive*, (4) holds. ■

Lemma 3.2. For all p and q the following holds invariantly.
(1) $S[p,q]>0 \Rightarrow f_p \neq nil \vee X[p] = active$,
(2) $f_p \neq nil \Leftrightarrow t_p < b_p$,
(3) $f_p \neq nil \Rightarrow R[f_p,p]>0$.

Proof. (1). Initially $S[p,q] = 0$. BS_p can only increase $S[p,q]$ if $x_p = active$, i.e. if $N[p] = active$. With lemma 3.1(4) we have that $N[p] = active$ implies $f_p \neq nil$ or $X[p] = active$. BR_p does not change $S[p,q]$ or $X[p]$, and if it changes f_p, it sets $f_p \neq nil$. BI_p and D_p change none of the variables involved. T_p sets $S[p,q] = 0$. Hence (1) holds.
(2). In BR_p f_p is set to $\neq nil$ if and only if b_p is moved to the end of Ω, and T_p sets $f_p = nil$ and moves t_p to the end of Ω. Hence (2) holds.
(3). Initially $f_p = nil$. If BR_p sets f_p to q, $R[q,p]$ was set to a value > 0, too. Other BR_p actions do not decrease $R[p,q]$. T_p sets $R[f_p,p]$ to 0, but sets $f_p = nil$ also. Hence (3) holds. ■

Lemma 3.3.
$f_p = nil \vee \{Q[f_p,p]+S[f_p,p]>0 \wedge (Q[f_p,p] \leq 0 \Rightarrow t_{f_p} < b_{f_p} < b_p \vee X[f_p] = active)\}$.

Proof. Initially $f_p = nil$. T_p sets $f_p = nil$. BI and D do not change any of the variables involved. BS_{f_p} does not decrease $S[f_p,p]$. Consider BR_p. Assume $f_p \neq nil$ beforehand. Then the second disjunct holds, and will not be violated by BR_p. Assume $f_p = nil$. If f_p is set to some value q, then by lemma 3.2, $R[q,p]>0$. Thus with lemma 3.1 $Q[q,p]+S[q,p]>0$, and hence $Q[q,p]>0$ or $S[q,p]>0$. If the former holds we are ready. If the latter holds, we have with lemma 3.2 $f_q \neq nil$ or $X[q] = active$. The former implies $t_q < b_q$, and as in the current BR_p action b_p is moved to the end of Ω, $t_q < b_q < b_p$ holds. If $f_p \neq nil$, we must consider actions BR_{f_p} and T_{f_p} also. If $f_{f_p} = nil$ before the BR_{f_p}

action, then $b_{f_p} < t_{f_p}$, and either $Q[f_p,p]>0$ or $S[f_p,p]>0$ and $X[f_p] = active$ holds. Neither of these statements is violated by BR_{f_p}. If $f_{f_p} \neq nil$ before the BR_{f_p} action, none of the variables involved is changed. T_{f_p} might set $X[f_p] = passive$, and sets $S[f_p,p] = 0$. As $Q[f_p,p]+S[f_p,p]>0$ beforehand, now $Q[f_p,p]>0$ holds. Thus the relation holds invariantly. ■

Theorem 3.4. $TERM_{tr} \vee \exists p,q\ Q[p,q]>0 \vee \exists p\ X[p] = active$.

Proof. Assume $TERM_{tr}$ does not hold. Then $\exists p,q\ M[p,q]>0$ or $\exists p\ N[p] = active$ holds. Let p and q be such that $M[p,q]>0$. Then $S[p,q]+Q[p,q]>0$. Thus $Q[p,q]>0$ or $X[p] = active$ or $f_p \neq nil$, by lemmas 3.1 and 3.2. On the other hand, let p be such that $N[p] = active$. Then $X[p] = active$ or $f_p \neq nil$. Assume $f_p \neq nil$. Let q be the process with the smallest b_q and $f_q \neq nil$. With lemma 3.3 we have that $X[f_q] = active$ or $Q[f_q,q]>0$, since $b_{f_q} > b_q$. ■

Lemma 3.5. $\forall p,q\ (\ Q[p,q]\geq 0 \vee X[p] = active \vee t_p < b_p < t_q\ )$.

Proof. Initially $Q[p,q] = 0$. BS, BI, and D do not change any of the variables involved. BR_p does not change $Q[p,q]$ or $X[p]$. If $t_p < b_p$ held beforehand, then $f_p \neq nil$, and the ordering Ω is not changed. T_p sets $S[p,q] = 0$, so $Q[p,q]\geq 0$ follows. T_q sets $R[p,q] = 0$, and $t_q > t_p$ and $t_q > b_p$. Thus $Q[p,q]\geq 0$ or $S[p,q]>0$. The latter implies $t_p < b_p$ or $X[p] = active$. ■

Theorem 3.6. $\exists p,q\ Q[p,q]>0 \Rightarrow \exists r\ \sum_s Q[s,r]>0 \vee \exists r\ X[r] = active$.

Proof. Let there be an entry $Q[p,q]>0$. If for all $u,v\ Q[u,v]\geq 0$, then $\sum_p Q[p,q]>0$. Thus assume there are some negative Q elements. Let s be the process such that $\exists x\ Q[x,s]<0$ and t_s minimal. Thus $S[x,s]>0$. Let r be the process such that $\exists y\ S[r,y]>0$ and b_r minimal. Then we know with lemma 3.5 that $t_x < b_x < t_s$. Moreover, $b_x \geq b_r$, and $f_r \neq nil$ or $X[r] = active$. Assume $f_r \neq nil$. Then $t_r < b_r$ and with lemma 3.3 we have $Q[f_r,r]>0$ or $X[f_r] = active$, since $S[f_r,r]>0$ and $b_{f_r} < b_r$ leads to a contradiction with the minimality of b_r. Since $t_r < b_r \leq b_x < t_s$, we know that $\forall x\ Q[x,r]\geq 0$. Hence $\sum_x Q[x,r]>0$. In all other cases there is an *active* X element. ■

Thus for detection purposes the values $\sum_q Q[q,p]$ suffice, and we can use the previous program skeleton if we define

$$L_p[p] = S[p,p] - \sum_q R[q,p], \quad \text{and} \quad L_p[q] = S[p,q] \text{ for } p \neq q.$$

This establishes the safety of the program skeleton of section 3.1.

4. Termination detection as distributed infimum approximation. Now we are going to write the program skeleton for termination detection in terms of distributed infimum approximation. Remember that termination detection can be considered as determining whether the infimum of all local values (of processes and messages) is *passive*, in the (partial) ordering where *passive* $\geq$ *active*. Let k be a fixed element from the partial ordering with which we

want to compare all local values. The formulation of the previous program skeleton then becomes:

Initially
$\forall p$: $X[p] = x_p \vee X[p] \not\geq k$, $f_p = nil$,
$\forall q$: $S[p,q] = 0$, $R[p,q] = 0$, $Q[p,q] = M[p,q] \geq 0$, and $b_p < t_q$ in Ω.

BS_p: **begin** send $<m, x_p>$ to q;
if $x_p \not\geq k$
then $S[p,q] := S[p,q]+1$ (* $M[p,q] := M[p,q]+1$ *)
fi
end

BR_p: {a message $<m,x>$ from q arrives at p}
begin receive $<m,x>$;
if $x \not\geq k$
then $R[q,p] := R[q,p]+1$; (* $M[q,p] := M[q,p]-1$ *)
$x_p := \inf(x, x_p)$; (* $N[p] := \inf(x, x_p)$ *)
if $f_p = nil$ **then** $f_p := q$; move b_p to the end of Ω **fi**
fi
end

BI_p: **begin** choose $x \geq x_p$; $x_p := x$ **end** (* $N[p] := x$ *)

T_p: {process p holds the token (Q,X)}
begin $f_p := nil$; move t_p to the end of Ω; $X[p] := x_p$;
forall q
do $Q[p,q] := Q[p,q]+S[p,q]$; $S[p,q] := 0$;
$Q[q,p] := Q[q,p]+R[q,p]$; $R[q,p] := 0$
od
end

D_p: {process p holds the token (Q,X)}
begin if $\forall r,s\ Q[r,s] \leq 0 \wedge \inf_p X[p] \geq k$ **then** *yield* (k) **fi end**

$TERM_{tr}$ = all messages underway are $\geq k$ and $\forall p\ x_p \geq k$,
or, equivalently,
$TERM_{tr}$ $= \forall p,q\ M[p,q] = 0 \wedge \inf_p N[p] \geq k$.

The formulation of the invariants is essentially the same, if we replace " = *passive*" by "$\geq k$" and " = *active*" by "$\not\geq k$". Likewise, we can reformulate "$\exists p\ X[p] =$ *active*" as "$\inf_p X[p] \not\geq k$". The reason for the notation of "$\not\geq k$" instead of "$< k$", is that in general posets this is not equivalent, as there can be incomparable elements.

Note however, that although we can correctly decide with this program skeleton whether the distributed infimum has reached the value k, this cannot be used as such for distributed infimum approximation. The reason is, that we want the token to reflect the continuously changing distributed infimum, and not just wait until it reaches some fixed value k. Hence we

need to change the program skeleton such that it handles all possible values of k simultaneously. Processes now have the problem to decide which messages to count: previously, only messages $\not\geq k$ were counted. As all values of k are a priori possible, processes now had better count all messages, and count separately for all different values such that the necessary information for some k can be extracted later on. The straightforward extension for, for example, the one counter $R[p,q]$ would be a set of counters $R^{\not\geq k}[p,q]$ for each possible value k. If there are many more possible values k than values actually used in messages, we can also use counters $R^{k}[p,q]$ = the number of messages q received from p with a value = k since the last token visit, only implementing counters with values $\neq 0$. Another possible implementation is using one multiset $R[p,q]$ which contains as many instances of the value k as there were messages from p received by q with a value of k. In this last case we need the convention that the multiset which represents the token matrix element $Q[p,q]$ can contain values negatively, corresponding to negative counters. If a value k is deleted from $Q[p,q]$ while it is not there, it is added as a negative value. If a value k is added (positively) to $Q[p,q]$ and $Q[p,q]$ contains this value negatively, then the negative value is deleted. Thus positive and negative values cancel each other, just as in the case of counters. If we want to consider the positive elements of a set $R[p,q]$ negatively and vice versa, we will write $-R[p,q]$. If we take the infimum over a multiset which contains elements negatively, we do not take the negative values into account.

4.1. Extended program skeleton for distributed infimum approximation. We will now rewrite the program skeleton of section 3.1 for all values of k simultaneously in terms of multisets. For ease of comparison, we state the corresponding statements in terms of counters $R^{\not\geq k}[p,q]$ in comment.

Initially
$\forall p$: $X[p] \leq x_p$,
  $\forall q$: $S[p,q] = \emptyset$, $R[p,q] = \emptyset$, $Q[p,q] = M[p,q]$,
    $\forall k$: $b_p < t_q$ in $\Omega^{\not\geq k}$, $f_p^{\not\geq k} = nil$.

BS_p: **begin** send $<m, x_p>$ to q;
    $S[p,q] := S[p,q] \cup \{x_p\}$ (* $M[p,q] := M[p,q] \cup \{x_p\}$ *)
    (* **forall** k **with** $x_p \not\geq k$ **do** $S^{\not\geq k}[p,q] := S^{\not\geq k}[p,q]+1$ **od** *)
  end

BR_p: {a message $<m,x>$ from q arrives at p}
  begin receive $<m,x>$;
    $R[q,p] := R[q,p] \cup \{x\}$; (* $M[q,p] := M[q,p] \cup -\{x\}$ *)
    (* **forall** k **with** $x \not\geq k$ **do** $R^{\not\geq k}[q,p] := R^{\not\geq k}[q,p]+1$ **od** *)
    $x_p := \inf(x, x_p)$; (* $N[p] := \inf(x, x_p)$ *)
    forall k **with** $x \not\geq k \wedge f_p^{\not\geq k} = nil$
    do $f_p^{\not\geq k} := q$; move b_p to the end of $\Omega^{\not\geq k}$ **od**
  end

BI_p: **begin choose** $x \geq x_p$; $x_p := x$ **end** (* $N[p] := x$ *)

T_p: {process p holds the token (Q,X)}
begin forall k **do** $f_p^{\geq k} := nil$; move t_p to the end of $\Omega^{\geq k}$ **od**;
$X[p] := x_p$;
forall q
do $Q[p,q] := Q[p,q] \cup S[p,q]$; $S[p,q] := \emptyset$;
$Q[q,p] := Q[q,p] \cup -R[q,p]$; $R[q,p] := \emptyset$
od
(* **forall** k
do **forall** q
do $Q^{\geq k}[p,q] := Q^{\geq k}[p,q] + S^{\geq k}[p,q]$; $S^{\geq k}[p,q] := 0$;
$Q^{\geq k}[q,p] := Q^{\geq k}[q,p] - R^{\geq k}[q,p]$; $R^{\geq k}[q,p] := 0$
od **od**
*)
end

D_p: {process p holds the token (Q,X)}
begin *yield* $(\inf(\inf_{r,s} \inf Q[r,s], \inf X))$
(* **if forsome** k $\forall r,s$ $Q^{\geq k}[r,s] \leq 0 \wedge \inf X \geq k$ **then** *yield* (k) **fi** *)
end

$k\text{TERM}_{tr}$ = all messages underway are $\geq k$ and $\forall p$ $x_p \geq k$,
or, equivalently,
$k\text{TERM}_{tr}$ = $\inf(\inf M, \inf N) \geq k$.

To simplify notation we write M, N, S, R, and X for $\bigcup_{p,q} M[p,q]$, $\bigcup_p \{N[p]\}$, $\bigcup_{p,q} S[p,q]$, $\bigcup_{p,q} R[p,q]$, and $\bigcup_p \{X[p]\}$, respectively.

4.2. Correctness proof for a fixed value of k. We begin by reformulating the lemmas and theorems of section 3.2 in terms of counters $R^{\geq k}[p,q]$ etc.. We will not give the corresponding proofs as they are completely analogous to those in section 3.2.

Lemma 4.1. For all p,q, and k the following holds invariantly.
(1) $R^{\geq k}[p,q] \geq 0$,
(2) $S^{\geq k}[p,q] \geq 0$,
(3) $Q^{\geq k}[p,q] + S^{\geq k}[p,q] = M^{\geq k}[p,q] + R^{\geq k}[p,q]$,
(4) $N[p] \geq k \vee X[p] \not\geq k \vee f_p^{\geq k} \neq nil$.

Lemma 4.2. For all p,q, and k the following holds invariantly.
(1) $S^{\geq k}[p,q] > 0 \Rightarrow f_p^{\geq k} \neq nil \vee X[p] \not\geq k$,
(2) $f_p^{\geq k} \neq nil \Leftrightarrow t_p < b_p$ in $\Omega^{\geq k}$,
(3) $f_p^{\geq k} \neq nil \Rightarrow R^{\geq k}[f_p^{\geq k}, p] > 0$.

Lemma 4.3.
$f_p^{\geq k} = nil \vee$
$\{Q^{\geq k}[f_p^{\geq k}, p] + S^{\geq k}[f_p^{\geq k}, p] > 0 \wedge (Q^{\geq k}[f_p^{\geq k}, p] \leq 0 \Rightarrow t_{f_p^{\geq k}} < b_{f_p^{\geq k}} < b_p \vee X[f_p^{\geq k}] \not\geq k)\}$.

Theorem 4.4. For all k the following holds invariantly.
$(\exists p,q\ M^{\not\geq k}[p,q]>0 \vee \exists p\ N[p]\not\geq k) \Rightarrow \exists p,q\ Q^{\not\geq k}[p,q]>0 \vee \exists p\ X[p]\not\geq k.$

Lemma 4.5. For all k the following holds invariantly.
$\forall p,q\ (Q^{\not\geq k}[p,q]\geq 0 \vee X[p]\not\geq k \vee t_p<b_p<t_q \text{ in } \Omega^{\not\geq k})$.

Theorem 4.6. For all k the following holds invariantly.
$\exists p,q\ Q^{\not\geq k}[p,q]>0 \Rightarrow \exists r \sum_s Q^{\not\geq k}[s,r]>0 \vee \exists r\ X[r]\not\geq k.$

Next we establish the relation between multisets and counters. We denote "is contained negatively in" by "$\in_{neg}$".

Lemma 4.7. For all p,q, and k we have
(1) $R^{\not\geq k}[p,q]>0 \Leftrightarrow \exists l$ with $(l\not\geq k \wedge l\in R[p,q]) \Leftrightarrow \inf R[p,q]\not\geq k$,
(2) $S^{\not\geq k}[p,q]>0 \Leftrightarrow \exists l$ with $(l\not\geq k \wedge l\in S[p,q]) \Leftrightarrow \inf S[p,q]\not\geq k$,
(3) $Q^{\not\geq k}[p,q]>0 \Leftrightarrow \exists l$ with $(l\not\geq k \wedge l\in Q[p,q]) \Leftrightarrow \inf Q[p,q]\not\geq k$,
(4) $Q^{\not\geq k}[p,q]<0 \Rightarrow \exists l$ with $(l\not\geq k \wedge l\in_{neg} Q[p,q])$,
(5) $M^{\not\geq k}[p,q]>0 \Leftrightarrow \exists l$ with $(l\not\geq k \wedge l\in M[p,q]) \Leftrightarrow \inf M[p,q]\not\geq k$,
(6) $\exists p\ N[p]\not\geq k \Leftrightarrow \inf N\not\geq k$,
(7) $\exists p\ X[p]\not\geq k \Leftrightarrow \inf X\not\geq k$.

Proof. Obvious from the program skeleton and properties of the partial ordering and infimum. ■

We can now rewrite the lemmas and theorems in terms of infimums of sets instead of counters. Note that for example $\inf \bigcup_p R[p,q] = \inf_p \inf R[p,q]$, whereas $\inf \bigcup_p Q[p,q]$ is not necessarily equal to $\inf_p \inf Q[p,q]$. This is the case because the multiset $Q[p,q]$ can contain elements negatively, and these elements could cancel other elements in the union $\bigcup_p Q[p,q]$ that contribute to the infimum $\inf_p \inf Q[p,q]$.

Lemma 4.8. For all p,q, and k the following holds invariantly.
(1) $R[p,q]$ contains no elements negatively,
(2) $S[p,q]$ contains no elements negatively,
(3) $Q[p,q] \cup S[p,q] = M[p,q] \cup R[p,q]$,
(4) $N[p]\geq k \vee X[p]\not\geq k \vee f_p^{\not\geq k} \neq nil$.

Proof. Combine the proof of lemma 4.1 with lemma 4.7. ■

Note that (3) implies that all elements which are contained negatively in $Q[p,q]$, occur in $S[p,q]$ and hence are canceled in the union.

Lemma 4.9. For all p,q, and k the following holds invariantly.
(1) $\inf S[p,q]\not\geq k \Rightarrow f_p^{\not\geq k} \neq nil \vee X[p]\not\geq k$,
(2) $f_p^{\not\geq k} \neq nil \Leftrightarrow t_p<b_p$ in $\Omega^{\not\geq k}$,
(3) $f_p^{\not\geq k} \neq nil \Rightarrow \inf R[f_p^{\not\geq k},p]\not\geq k$.

Proof. Combine the proof of lemma 4.2 with lemma 4.7. ■

Lemma 4.10. For all p and k the following holds invariantly.
$f_p^{\not\geq k} = nil \vee \{\inf(Q\,[f_p^{\not\geq k},p] \cup S\,[f_p^{\not\geq k},p]) \not\geq k \wedge$
$(\inf Q\,[f_p^{\not\geq k},p] \geq k \Rightarrow t_{f_p^{\not\geq k}} < b_{f_p^{\not\geq k}} < b_p \text{ in } \Omega^{\not\geq k} \vee X\,[f_p^{\not\geq k}] \not\geq k)\}$.

Proof. Combine the proof of lemma 4.3 with lemma 4.7. ■

Theorem 4.11. For all k the following holds invariantly.
$\inf(M \cup N) \not\geq k \Rightarrow \exists p,q \inf Q\,[p,q] \not\geq k \vee \inf X \not\geq k$.

Proof. Combine the proof of theorem 4.4 with lemma 4.7. ■

Lemma 4.12. For all p,q, and k the following holds invariantly.
$\exists l \in_{neg} Q\,[p,q]$ with $l \not\geq k \Rightarrow X\,[p] \not\geq k \vee t_p < b_p < t_q$ in $\Omega^{\not\geq k}$.

Proof. Combine the proof of lemma 4.5 with lemma 4.7. ■

Theorem 4.13. For all k the following holds invariantly.
$\exists p,q \inf Q\,[p,q] \not\geq k \Rightarrow \exists r \inf \bigcup_s Q\,[s,r] \not\geq k \vee \exists r\, X\,[r] \not\geq k$.

Proof. Combine the proof of theorem 4.6 with lemma 4.7. ■

4.3. Distributed infimum approximation. The consequence of theorems 4.11 and 4.13 is that we now know which value to take as an approximation of the sought infimum.

Theorem 4.14. $\inf(\inf X, \inf_p \inf \bigcup_q Q\,[q,p]) \leq \inf(M \cup N)$.

Proof. Take $k = \inf(\inf X, \inf_p \inf \bigcup_q Q\,[q,p])$ and use theorems 4.11 and 4.13. ■

Hence the value $\inf(\inf X, \inf_p \inf \bigcup_q Q\,[q,p])$ is a continuous approximation of the distributed infimum $\inf(M \cup N)$. We will now rewrite the program skeleton for distributed infimum approximation with multisets, where all features that were only necessary for the proof are left out. As all references to the value k will disappear, this is indeed a program skeleton suitable for distributed infimum approximation.

Initially
$\forall p$: $X\,[p] \leq x_p$, $Q\,[p] = \{x \mid <m,x> \text{ initially underway to } p\}$,
$\forall q$: $L_p[q] = \varnothing$.

BS_p: **begin** send $<m,x_p>$ to q; $L_p[q] := L_p[q] \cup \{x_p\}$ **end**

BR_p: {a message $<m,x>$ arrives at p}
begin receive $<m,x>$; $L_p[p] := L_p[p] \cup - \{x\}$; $x_p := \inf(x,x_p)$ **end**

BI_p: **begin** choose $x \geq x_p$; $x_p := x$ **end**

T_p: {process p holds the token (Q,X)}
begin forall q **do** $Q\,[q] := Q\,[q] \cup L_p[q]$; $L_p[q] := \varnothing$ **od**;
$X\,[p] := x_p$
end

D_p: {process p holds the token (Q, X)}
begin *yield* $(\inf(\inf X, \inf_p \inf Q[p]))$ **end**

Thus it is indeed possible to transform a termination detection algorithm to a distributed infimum approximation algorithm together with its assertional correctness proof.

5. Conclusions. We proposed a new and fairly complex algorithm for distributed infimum approximation. The algorithm was obtained and its safety verified by systematically deriving it from a termination detection algorithm for the so-called transactional model of computation. To complete the development of the algorithm we will briefly sketch four options for the organization of token-visits together with their liveness properties (section 5.1). Related work will be discussed in section 5.2.

5.1. Token tour, liveness, and message complexity. What mechanisms can direct the token visits in such a way that liveness of the resulting algorithm is guaranteed while the message complexity of the algorithm remains as low as possible? We briefly discuss four options for such a mechanism: a predefined tour, movements based upon information in the token, movements on request of processes, and a token as a process on a fixed location.

The easiest mechanism is to define a cyclic tour through all processes and send the token along this tour repeatedly. This solution does not exploit the main advantage of our scheme (as explained in section 2.5) because it implements a repeated observation of all processes. Liveness of this solution is easy to demonstrate. When the token has completed a full tour after F has reached a value $\geq k$, all values $\not\geq k$ have disappeared from the token and the next D operation yields an approximation k' with $k' \geq k$. Unfortunately, an infinite number of token-visits can occur without any progress being made.

The efficiency of this scheme can be improved by using information in the token. The token is only sent to processes with the property that a change in the information of this process possibly results in an increase of the approximation included in the token. If the ordered set A contains two incomparable elements however, it is still the case that an infinite number of token moves can occur without any progress being made, due to the liveness requirement. Let p and q be the only two processes, x_p and x_q be incomparable, $F = \inf(x_p, x_q)$, and no messages are underway. An internal action in p could increase x_p to $\sup(x_p, x_q)$, which would increase F to x_q, and similarly an internal action in q could increase F to x_p. Thus the token is forced to visit both p and q infinitely often even if no further basic actions take place.

If however the ordering in A is total, we can refine the mechanism further so that a process holds the token until its own component in the token contains no negative entries and has a higher infimum than the token as a whole. By the totality of A the mechanism still satisfies liveness, and now the number of token moves is bounded by the number of operations in the basic computation. This is used especially in termination detection.

A possibility to reduce the number of token-visits in the above mechanism for non-total A is to send the token to a process only on its request. A process requests the token only if it has an update to make to its contents. This again bounds the number of token-visits to the number of operations in the basic computation, but now (due to the unpredictability of the token location) the request messages become the complexity bottleneck.

Finally, it is possible to let the "token" reside in a fixed process, and have the processes send their updates to the token in a message. In this scheme the token is viewed as a process rather than as a wandering message. In this case we must take care that the updates are processed in the correct order. The fault-tolerance of this scheme can be improved by having several copies of the token process as in [HJ87].

Concluding this section, we see that determining the token moves and obtaining liveness is a more delicate question in the new algorithm than it was in the underlying termination detection algorithm. If the new algorithm should be used in practice, efforts should be made to evaluate the message complexity of the sketched mechanisms in an analytical or empirical manner.

5.2. Related publications. The problem of distributed infimum approximation was first posed in [Te86] and its treatment was further developed in [Te89]. As noted already in [Te86], the problem generalizes the problem of termination detection, but more direct applications of its solutions are found in [Hu85] and [Je85]. [Kl43] is an early, axiomatic treatment of the used concepts poset and infimum.

The relation between the distributed infimum approximation problem and the termination detection problem was further developed in [Te89]. It was shown that the verification of (the safety of) a distributed infimum approximation algorithm can be done by verifying a suitable derived termination detection algorithm. The conclusion is, that all of the essential difficulties of the distributed infimum approximation problem are somehow present in the problem of termination detection already. This poses the question whether a general termination detection algorithm can be "upgraded" to to an algorithm for distributed infimum approximation.

We chose to try this out with a termination detection algorithm by Mattern [Ma87], because this algorithm shows interesting properties that are not obtained by the known algorithms in [Te89]. Earlier [ST88] we developed a correctness proof for this algorithm using the method of system-wide invariants, as advocated in [Kr78], [Kn81], and [Te89]. A similar termination detection algorithm was proposed by Hélary et al. [HJ87], who introduced the idea of implementing the token as a process rather than as a message. The algorithm is based on snapshots that are collected by the token, but these are not necessarily consistent in the sense of [LY87]. It was however shown in [LY87] that termination is *strongly stable*, that is, it can be detected even using inconsistent snapshots. This paper generalizes the results of [Ma87], [ST88], [LY87], and [HJ87] to the problem of distributed infimum approximation.

6. References.

[HJ87] Hélary, J.-M., C. Jard, N. Plouzeau, and M. Raynal, *Detection of stable properties in distributed systems*, Proc. 6th PoDC, Vancouver, Canada, 1987, pp. 125-136.

[Hu85] Hughes, J., *A distributed garbage collection algorithm*, in: J.P. Jouannoud (ed.), Functional programming languages and computer architecture, LNCS 201, Springer Verlag, Heidelberg, pp. 256-272.

[Je85] Jefferson, D., *Virtual Time*, ACM ToPLaS 7 (1985), 404-425.

[Kl43] Klein-Barmen, F., *Über gewisse Halbverbände und kommutative Semigruppen, Teil I*, Mathematische Zeitschrift 48 (1943), 275-288.

[Kn81] Knuth, D.E., *Verification of Link-Level Protocols*, BIT 21 (1981), 31-36.

[Kr78] Krogdahl, S., *Verification of a Class of Link-Level Protocols*, BIT 18 (1978),

436-448.

[LY87] Lai, T.H., and T.H. Yang, *On distributed snapshots*, Inf. Proc. Lett. 25 (1987), 153-158.

[Ma87] Mattern, F. *Algorithms for distributed termination detection*, Distributed Computing 2 (1987), 161-175.

[ST88] Schoone, A.A., and G. Tel, *Assertional verification of a termination detection algorithm*, Techn. Rep. RUU-CS-88-6, Dept. of Computer Science, University of Utrecht, Utrecht, 1988.

[Te86] Tel, G., *Distributed infimum approximation*, Techn. Rep. RUU-CS-86-12, Dept. of Computer Science, University of Utrecht, Utrecht, 1986.

[Te89] Tel, G., *The structure of distributed algorithms*, Ph.D. Thesis, Utrecht, 1989.