A Termination Detection Algorithm: Specification and Verification

Robert Eschbach

Department of Computing Sciences,
University of Kaiserslautern, PO 3049
D-67653 Kaiserslautern, Germany

Abstract. We propose a methodology for the specification and verification of distributed algorithms using Gurevich's concept of Abstract State Machines. The methodology relies on a distinction between a higher-level specification and a lower-level specification of an algorithm. The algorithm is characterized by an informal problem description. A justification assures the appropriateness of the higher-level specification for the problem description. A mathematical verification assures that the lower-level specification implements the higher-level one and is based on a refinement-relation. This methodology is demonstrated by a well-known distributed termination detection algorithm originally invented by Dijkstra, Feijen, and van Gasteren.

1 Introduction

In this paper we propose a methodology for the specification and verification of distributed algorithms using Gurevich's concept of Abstract State Machines (cf. [Gur95], [Gur97], [Gur99]). The development of distributed algorithms usually starts with an informal *problem description* (see figure 2). In order to get a mathematical model of the problem description at the starting point of construction one has to choose what often is called a ground model (cf. [Bör99]) or a *higher-level specification* for the problem description. In this paper the higher-level specification is an Abstract State Machine (ASM) and as such it constitutes a well-defined mathematical object. An *informal justification*[1] shows the appropriateness of the higher-level specification for the problem description (cf. [Bör99]). A so-called *lower-level specification* represents the algorithm on a more concrete abstraction level as the higher-level specification. The *mathematical verification* guarantees that the lower-level specification implements the higher-level specification and is usually based on refinement relations. In this paper we focus mainly on the mathematical verification.
We use a well-known distributed algorithm, namely a *termination detection algorithm* originally invented by Dijkstra, Feijen and van Gasteren in [DFvG83] as an example to show how such an algorithm can be specified and verified within this methodology using Abstract State Machines. We give in this paper

[1] Since the problem description is informal, a mathematical proof is not possible.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1720–1737, 1999.
© Springer-Verlag Berlin Heidelberg 1999

a *correctness proof* for a variation[2] of the algorithm presented in [DFvG83]. As in [BGR95] our correctness proof relies on a distinction between a higher-level view and a lower-level view of the algorithm. The proof itself is given on a detailed mathematical level using thereby standard techniques from mathematics like case distinction or induction. We introduce for both the higher-level specification and the lower-level specification a kind of *stuttering steps*. The concept of stuttering is well-known from TLA (cf. [Lam94]). Stuttering steps represent steps in which a machine changes its local state. These changes are invisible. The stuttering itself leads to a simple and natural refinement relation which eases the construction of our correctness proof.

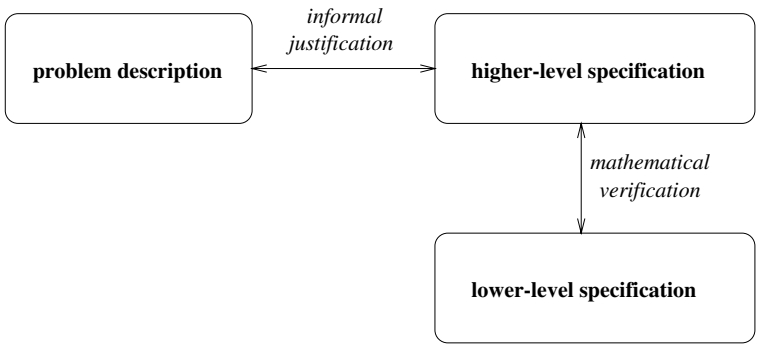

Fig. 1. Underlying Methodology

We specify and verify the termination detection algorithm of Dijkstra, Feijen, van Gasteren on a detailed mathematical level. The reader who is interested in a more intuitive explanation and an excellent derivation of this algorithm is referred to [Dij99] (or [DFvG83] for the original version). We start with a description of the *problem of termination detection* [Dij99]:

> We consider N machines, each of which is either active or passive. Only active machines send what are called "messages" to other machines; each message sent is received some finite period of time later. After having received a message a machine is active; the receipt of a message is the only mechanism that triggers for a passive machine its transition to activity. For each machine, the transition from the active to the passive state may occur "spontaneously". From the above it follows that the state in which

[2] The variation itself stems from Shmuel Safra. The variation is that message transmission no longer needs to be instantaneous and is described in [Dij99]. In [Dij99] Safra's algorithm is derived along the very same lines as in [DFvG83]. Note also that in [DFvG83] the authors present a [DS80]-algorithm for the detection of the termination of a distributed computation.

> all machines are passive and no messages are on their way is stable: the distributed computation with which the messages are associated is said to have terminated. The purpose of the algorithm to be designed is to enable one of the machines, machine nr. 0 say, to detect that this stable state has been reached.

We denote the process by which termination is detected as "the probe". In addition to messages, machines can send what are called "signals" to other machines. We adopt a circular arrangement of the machines, more precisely, we assume that machine nr. 0 can send a signal to machine nr. N-1 and that machine nr. i+1 can send a signal to machine nr. i. Note that a machine can send a signal irrespective of its activity state. Especially this means that a passive machine can send a signal but cannot send a message.

This paper is organized as follows. In section 2 we construct the higher-level specification $\mathcal{A}$. This ASM represents the problem of termination detection stated above. We present domains, functions, modules, runs, and constraints of $\mathcal{A}$. In section 3 we construct the lower-level specification $\mathcal{A}'$. This ASM represents the termination detection algorithm presented in [Dij99]. In the lower-level specification the probe is implemented by adding new rules and refining old ones of $\mathcal{A}$, respectively. We present the lower-level specification in the same way as the higher-level one. Section 4 presents the correctness proof for the termination detection algorithm. First we define what it means for the lower-level ASM $\mathcal{A}'$ to implement the higher-level ASM $\mathcal{A}$. Then we prove that $\mathcal{A}'$ implements $\mathcal{A}$. In section 5 we give some concluding remarks.
Throughout this paper we assume the reader to be familiar with Gurevich's ASMs, especially with distributed ASMs, cf. [Gur95].

2 Higher-Level Specification

This section presents a higher-level specification for the problem description given in the introduction. The higher-level specification is given as a distributed ASM. In the following we describe a distributed ASM by its

1. domains,
2. functions,
3. modules,
4. runs,
5. constraints.

Domains (i.e., sets) can be classified into static and dynamic domains, i.e., domains which are changeable during a run or not changeable, respectively. *Functions* can be classified into internal, shared, and external functions. Our classification is based on a broad distinction between ASM agents and the environment. A more detailed classification can be found in [Bör99]. Internal functions can be changed by ASM agents only. A shared function can be affected by both ASM

agents and the environment. External functions can be changed by the environment only. Furthermore functions can be classified into static or dynamic functions, i.e., functions which are changeable during a run or not changeable, respectively. *Modules* are ASM rules (programs) which are associated with ASM agents. In a run an agent executes its associated module. In this paper a *run* is essentially an infinite sequence of states S_k and an infinite sequence of ASM agents A_k such that S_k can be transformed by agent A_k and the environment into state S_{k+1}. *Constraints* can be used to impose conditions upon functions, e.g. external functions.

2.1 Domains of $\mathcal{A}$

We define `Machine` to be the static universe of machine identifiers $\{0, \ldots, N-1\}$. We assume each machine identifier to be associated with an agent in the distributed ASM $\mathcal{A}$. In the following instead of agents we simply speak of machines of $\mathcal{A}$. `Bool` denotes the domain $\{\text{true}, \text{false}\}$ of boolean values, `Nat` the universe of natural numbers, and `Int` the universe of integers. The set `{SM, RM, P, S}` represents the set of so-called execution-modes.

2.2 Functions of $\mathcal{A}$

Let τ be the *vocabulary* of $\mathcal{A}$. Besides some standard functions on `Bool`, `Nat`, and `Int` the vocabulary τ is defined by the following functions. Messages are realized by an internal, dynamic function

```
messages : Machine -> Nat.
```

We assume that initially `messages` has value `0` for all machines. A machine can send a message (SM), receive a message (RM), execute the probe (P), or execute a skip (S). For this purpose we introduce an external, dynamic function

```
mode : Machine -> {SM,RM,P,S},
```

which determines the execution mode for each machine. A machine can either be active or be passive. The shared, dynamic function

```
active: Machine -> Bool,
```

determines the activity state for each machine. Active machines can send messages to other machines. The external, dynamic function

```
receivingMachine: -> Machine
```

determines the destination of a message transmission. In order to detect termination we introduce the external, dynamic function

```
terminationDetected : -> Bool.
```

We assume `terminationDetected` initially to be `false`.

2.3 Modules of $\mathcal{A}$

Each machine executes a module consisting of the rules SendMessage, ReceiveMessage, and Skip.
Sending a message to machine `j` is realized by incrementing `messages(j)`, receiving a message by machine `i` by decrementing `messages(i)`. Note that only active machines can send messages.

SendMessage

```
if mode(me) = SM and active(me) = true then
   messages(receivingMachine) := messages(receivingMachine) + 1
```

On receipt of a message, the receiving machine becomes active. Note that machines can receive messages irrespective of their activity state.

ReceiveMessage

```
if mode(me) = RM and messages(me) > 0 then
   messages(me) := messages(me) - 1, active(me) := true
```

The rule Skip realizes stuttering steps, i.e., steps in which machines invisibly change their local states. In $\mathcal{A}$ the probe is specified by execution-mode `P`, rule Skip, and the external, dynamic function `terminationDetected` constrained by properties given in section 2.5. An executing machine in mode `P` performs a stuttering step. Stuttering steps in a run can be replaced by concrete transitions in the lower-level specification. In this way we obtain a simple and natural refinement relation.

Skip

```
if mode(me) = S or mode(me) = P then skip
```

2.4 Runs of $\mathcal{A}$

We rely on the notion of *partially ordered runs* of [Gur95] generalized to external and shared functions and specialized to the linear ordered set of moves $(\mathrm{Nat}, <)$. We consider only infinite runs. Since the only agents in $\mathcal{A}$ are machines, the function `A` which determines for each $k \in \mathrm{Nat}$ an agent performing move k, is a mapping from the natural numbers to machine identifiers. The state function `S` is a mapping from the natural numbers to the states of $\mathcal{A}$. We define a *possible run* of $\mathcal{A}$ to be a tuple (A, S) with $A : \mathrm{Nat} \to \mathrm{Machine}$ and $S : \mathrm{Nat} \to \mathrm{State}$ such that

1. $S(0)$ is an initial state of $\mathcal{A}$, i.e., $S(0)$ fulfills the initial conditions given in section 2.2, and

2. state $S(k+1)$ is obtained from $S(k)$ by executing the module of machine $A(k)$ at $S(k)$ and then executing an action of the environment, i.e. by changing shared and external functions in an arbitrary way.

Instead of $S(k)$ (or $A(k)$) we sometimes write S_k (or A_k). In the following we simply say S_{k+1} is obtained from S_k by executing machine A_k, i.e., the action of the environment is omitted. We speak of move k or of step $(k, k+1)$ in run ρ. We say ρ is a *run* of $\mathcal{A}$ if ρ satisfies the constraints given in the following section 2.5.

Remark. We could use real-time semantics with either instantaneous or durative actions, like in [BGR95], but the algorithm can be formulated naturally and adequately in this simpler semantics.

2.5 Constraints of $\mathcal{A}$

We present the constraints of $\mathcal{A}$ using the standard temporal operators $\Box$ and $\Diamond$. We denote the value that a term t takes at time k in a run ρ by t_k. In the following let $\rho = (A, S)$ be a possible run of $\mathcal{A}$.

C0: $\forall i \in \text{Machine} : \Box\Diamond(A = i \wedge \text{mode}(i) = \text{RM})$

Intuitively, constraint C0 assures that a message sent is received a finite period of time later. Note that a machine may never send a message but must receive a sent message. Suppose one of the machines sends machine i a message. Thus there exists a time point, say k, with $messages(i)_k > 0$. From C0 we know there exists a time point, say l, with $k \leq l$ such that machine i is executed in step $(l, l+1)$ with $mode(i)_l = \text{RM}$. In this step machine i receives a sent message.

C1: The environment can change the function `active` in step $(k, k+1)$ only from true to false and this only if $\text{mode}(\text{A}_k)_k \neq \text{RM}$.

The receipt of a message is the only way for a passive machine to become active. Constraint C1 guarantees that the environment cannot change the activity state of an active machine which tries to receive a message. This avoids "inconsistencies" between receipt of a message and "spontaneous" transitions from the active to the passive state by the environment. Note that constraints C0 and C1 ensure a kind of "fair" runs. The ASM $\mathcal{A}$ together with C0 and C1 model the distributed system in the problem description.

Let $B_k = \sum i : 0 \leq i < N : \text{messages}(i)_k$ denote the number of messages on their way at time k. Termination at time k can then be characterized by:

$$\text{Termination}_k \triangleq B_k = 0 \wedge (\forall i \in \text{Machine} : \text{active}(i)_k = false).$$

Now we pose some constraints on terminationDetected. These constraints are an essential part of the higher-level specification.

C2: $\Box$(terminationDetected $\rightarrow$ Termination)

Constraint C2 assures that the value of terminationDetected is correct, i.e., if termination is detected in ρ then there is termination.

C3: $\Box$(Termination $\rightarrow \Diamond$ terminationDetected)

Constraint C3 makes sure that if there is termination in ρ then terminationDetected eventually becomes true.

C4: $\Box$(terminationDetected $\rightarrow \Box$ terminationDetected)

Constraint C4 guarantees that terminationDetected remains true if it is once set to true. Constraints C2, C3, and C4 ensure "good" runs, i.e., runs in which the value of terminationDetected is true. Constraints C2, C3, and C4 essentially specify correctness of the termination detection problem. Note that constraints C3 and C4 implies $\Box$(Termination $\rightarrow \Diamond\Box$ terminationDetected). The converse implication does not hold.

Remark. We think the higher-level specification $\mathcal{A}$ represents faithfully the problem description given in section 1 and hence can be seen as an adequate mathematical model of the informal problem description. Nevertheless a justification for this ASM is necessary. We refer the reader to [Bör99] for further information on the problem of justification. We concentrate in this paper mainly on the mathematical verification. A detailed justification is beyond the scope of this paper.

3 Lower-Level Specification

This section presents a lower-level specification for the algorithm presented in [DFvG83]. The lower-level specification is given as a distributed ASM. We describe $\mathcal{A}'$ in the same way as $\mathcal{A}$.

3.1 Domains of $\mathcal{A}'$

The abstract state machine $\mathcal{A}'$ has the same domains as $\mathcal{A}$.

3.2 Functions of $\mathcal{A}'$

Let τ' be the *vocabulary* of $\mathcal{A}'$. We do not mention standard functions on `Bool`, `Nat`, and `Int`. Vocabulary τ is a subvocabulary of τ', i.e., $\tau \subseteq \tau'$. The following functions of $\mathcal{A}'$ coincides on declaration, classification and initial conditions with the ones given in $\mathcal{A}$.

1. `messages : Machine -> Nat` (internal, dynamic)
2. `mode : Machine -> {SM,RM,P,S}` (external, dynamic)
3. `active: Machine -> Bool` (shared, dynamic)
4. `receivingMachine: -> Machine` (external, dynamic)

In $\mathcal{A}$ the function `terminationDetected` is an *external*, dynamic function. In $\mathcal{A}'$ the function `terminationDetected` is an *internal*, dynamic function.

```
terminationDetected : -> Bool.
```

We assume `terminationDetected` initially to be `false`.
Now we present new functions of $\mathcal{A}'$, i.e., functions of $\mathcal{A}'$ which are not part of the signature of $\mathcal{A}$. Each machine has a local message counter which is modeled by an internal dynamic function

```
c: Machine -> Int.
```

This local message counter can be incremented (sending a message) or decremented (receiving a message) by each machine. The intention is that counter `c(i)` represents local knowledge which can be used and changed only by machine i, i.e., `c(i)` can be seen as an internal location for machine i. We assume that initially all local message counters have value `0`.
Each machine can turn either white or black. The color of a machine is realized by an internal, dynamic function

```
color: Machine -> {black, white}.
```

We assume each machine initially to be white.
We describe the probe as a token being sent around the ring using signalling facilities. The token is realized by an internal, dynamic function

```
token: Machine -> Bool.
```

Initially `token` is `false` for all machines. Like machines the token can turn either white or black. Since there is at most one token propagated through the ring we use nullary, internal, dynamic functions to model color and value of the token, respectively.

```
tokenColor: -> {black, white}
tokenValue: -> Int
```

The function `tokenColor` is assumed initially to be `white` and the function `tokenValue` initially to be `0`. Note that M_0 initiates the probe by transmitting the token to M_{N-1} and each M_{i+1} transmits the token to M_i. We use the internal, static function

```
next: Machine -> Machine,
```

to model this circular arrangement of machines[3].
The internal, static function

```
id: Machine -> Machine,
```

returns for each machine its machine identifier.
We assume that machine M_0 can initiate the probe. We realize the initiation of the probe by a shared function

```
initiateProbe : -> Bool.
```

We assume `initiateProbe` initially to be `false`.

3.3 Modules of $\mathcal{A}'$

Each machine executes a module consisting of the rules SendMessage, ReceiveMessage, TransmitToken, InitiateProbe, NextProbe, and Skip.
Rule SendMessage of $\mathcal{A}'$ is a refined version of the one of $\mathcal{A}$. In the refined version additionally the local message counter c(me) is affected.

SendMessage

```
if mode(me) = SM and active(me) = true then
   messages(receivingMachine) := messages(receivingMachine) + 1,
   c(me) := c(me) + 1
```

On receipt of a message, the receiving machine is active and turns black. Note that machines can receive messages irrespective of their activity state. Note further that rule ReceiveMessage is a refined version of the one of $\mathcal{A}$ since it coincides with the latter on vocabulary τ.

ReceiveMessage

```
if mode(me) = RM and messages(me) > 0 then
   messages(me) := messages(me) - 1, active(me) := true,
   c(me) := c(me) - 1, color(me) := black
```

[3] $\text{next}(0) = N-1$, $\text{next}(N-1) = N-2, \ldots, \text{next}(1) = 0$

In $\mathcal{A}$ the probe is realized by stuttering steps (rule Skip) and the external, dynamic function `terminationDetected` constrained by C2, C3, C4. In $\mathcal{A}'$ the probe is realized by the rules TransmitToken, InitiateProbe and NextProbe and the *internal*, dynamic function `terminationDetected`. Upon token transmission[4] the executing machine is passive and turns white. The value of the token is changed according to the local message counter. Note also that in the following rule only machines M_{i+1} can transmit the token.

TransmitToken

```
if mode(me) = P and token(me) = true and
   active(me) = false and id(me) <> 0 then
   token(me) := false, token(next(me)) := true,
   if color(me) = black then tokenColor := black,
   tokenValue := tokenValue + c(me), color(me) := white
```

If machine 0 is executed in mode P and $initiateProbe = true$ holds then a new token is created and transmitted to machine $N-1$. The token itself is white and has value 0. Furthermore machine 0 turns to white and sets `initiateProbe` to `false`. The latter avoids "nested" probes.

InitiateProbe

```
if mode(me) = P and id(me) = 0 and initiateProbe = true then
   token(next(me)) := true, tokenValue := 0,
   tokenColor := white, color(me) := white,
   initiateProbe := false
```

At token return machine 0 investigates whether the stable state of termination has been reached or not. We say, the probe has been *successful* if at token return `c(0) + tokenValue = 0`, machine 0 is white and passive, and the token is white. After an successful probe machine 0 sets `terminationDetected` to `true`. After an unsuccessful probe machine 0 initiates a next probe by setting `initiateProbe` to `true`.

NextProbe

```
if mode(me) = P and id(me) = 0 and token(me) = true then
   if c(me) + tokenValue = 0 and color(me) = white and
      tokenColor = white and active(me) = false then
      terminationDetected := true
   else
      initiateProbe := true, token(me) := false
```

[4] via signal communication facilities which are available irrespective of facilities for message transmission, cf. section 1

As in $\mathcal{A}$ we realize stuttering by a rule Skip. Note also that in $\mathcal{A}$ an executing machine in mode P performs a stuttering step (cf. rule Skip of $\mathcal{A}$).

SKIP

```
if mode(me) = S then skip
```

3.4 Runs of $\mathcal{A}'$

We use the same notion of *possible run* of $\mathcal{A}'$ as for $\mathcal{A}$. We say ρ' is a *run* of $\mathcal{A}'$ if ρ' satisfies constraints C0, C1, and the constraints given in the following section 3.5.

3.5 Constraints of $\mathcal{A}'$

In the following let $\rho' = (A', S')$ be a possible run of $\mathcal{A}'$. Besides constraints C0 and C1 (stated for $\mathcal{A}'$) a run of $\mathcal{A}'$ has to fulfill the following constraints.

C5: $\forall i \in \text{Machine} : \Box\Diamond(\text{A}' = i \wedge \text{mode}(i) = P)$

Intuitively, constraint C5 assures that token transmission proceeds. More precisely, constraint C5 assures that each machine is executed infinite many times in mode P.

C6: The environment sets initiateProbe in ρ' exactly once to true.

Intuitively, constraint C6 assures that in each run the first probe is initiated by the environment. Note that initially initiateProbe is false and cannot changed by a machine until it is set to true by the environment.

4 Verification

In this section we show that the lower-level ASM $\mathcal{A}'$ is an implementation of $\mathcal{A}$. In the first subsection we define what it means for $\mathcal{A}'$ to implement $\mathcal{A}$. In the second subsection we prove that $\mathcal{A}'$ implements $\mathcal{A}$.

4.1 Implementation

In this subsection we define what it means for the lower-level ASM $\mathcal{A}'$ to *implement* the higher-level ASM $\mathcal{A}$.
There exists two distinct approaches to specification which Lamport calls in [Lam86] the *prescriptive* and *restrictive* approaches. In the prescriptive approach an implementation must exhibit all the same possible behaviors as the specification. In the restrictive approach, it is required that every possible lower-level behavior is represented by a higher-level behavior. In this paper the intention is

that the lower-level specification $\mathcal{A}'$ should satisfy constraints C2, C3, C4, i.e., each run of $\mathcal{A}'$ should satisfy C2, C3, C4. We do not require that all higher-level runs are implemented in a single implementation. Otherwise an implementation has to detect termination in the moment it occurred (cf. constraint C3). We adopt here the restrictive approach.
As in first-order logic, the *reduct* of an τ'-state S' to the vocabulary τ is the state S denoted by $S'|\tau$ obtained from S' by restricting the interpretation of function names on τ' to τ. Note that τ is a subvocabulary of τ'.
Now we define a *refinement-relation*, more precisely, we define a refinement-relation between $\mathcal{A}$ and $\mathcal{A}'$ which is sufficient to give the corresponding correctness proof. We say a run $\rho' = (A', S')$ of $\mathcal{A}'$ *implements* a run $\rho = (A, S)$ of $\mathcal{A}$ if

1. $S_k \cong S'_k|\tau$, and
2. $A_k = A'_k$

for all $k \in$ Nat. Call a run ρ of $\mathcal{A}$ a *higher-level run* and a run ρ' of $\mathcal{A}'$ a *lower-level run*, respectively. We say that ASM $\mathcal{A}'$ *implements* $\mathcal{A}$ iff each lower-level run implements a higher-level run. If $\mathrm{Run}(\mathcal{A})$ denotes the collection of runs of $\mathcal{A}$ and $\mathrm{Run}(\mathcal{A}')$ the collection of runs of $\mathcal{A}'$, respectively, then this refinement-relation can be characterized by

$$\mathrm{Run}(\mathcal{A}')|\tau \subseteq \mathrm{Run}(\mathcal{A}),$$

where $\mathrm{Run}(\mathcal{A}')|\tau$ denotes the collection of runs of $\mathcal{A}'$ restricted to the vocabulary τ of $\mathcal{A}$ (cf. the construction of ρ in the following section 4.2). Look at [AL91] for a detailed discussion under which assumptions the existence of a refinement-relation can be guaranteed.

4.2 $\mathcal{A}'$ Implements $\mathcal{A}$

Now we will prove that $\mathcal{A}'$ implements $\mathcal{A}$. We have to show that each lower-level run is an implementation of a higher-level run. Let $\rho' = (A', S')$ be an arbitrary run of $\mathcal{A}'$. We define a tuple $\rho = (A, S)$ by:

1. $A_k := A'_k$
2. $S_k := S'_k|\tau$

We show that ρ is a run of $\mathcal{A}$. We make the following observations: (i) constraints C0 and C1 are satisfied in ρ (denoted by $\rho \models C0 \wedge C1$), (ii) $S_k \cong S'_k|\tau$ and $A_k = A'_k$ hold for all $k \in$ Nat. It remains to show that (i) ρ is a possible run of $\mathcal{A}$, and (ii) $\rho \models C2 \wedge C3 \wedge C4$. In this case ρ is a run of $\mathcal{A}$. From this we can conclude that ρ' is an implementation of ρ and hence that $\mathcal{A}'$ implements $\mathcal{A}$.
Since $\mathcal{A}$ and $\mathcal{A}'$ require the same initial conditions for functions from τ we know that $S'_0|\tau$ is an initial state of $\mathcal{A}$. It remains to show that S_{k+1} can be obtained from S_k by executing machine A_k in $\mathcal{A}$, i.e., that the so-called reduct property depicted in Fig. 2 holds. Note that in $\mathcal{A}'$ (and in $\mathcal{A}$) for each time point at most one guard of the rules is satisfied.

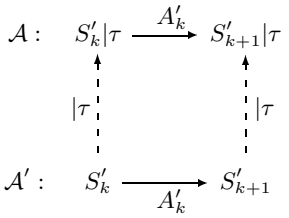

Fig. 2. Reduct Property

Lemma 1. *For all* $k \in$ Nat *state* S_{k+1} *can be obtained from* S_k *by executing machine* A_k *in* $\mathcal{A}$*.*

Proof. Let $k \in$ Nat. We simply say R_k holds, if the reduct property holds for k, i.e. if $S'_{k+1}|\tau$ is obtained from $S'_k|\tau$ by executing machine A'_k. (1) Assume $\text{mode}(\text{A}_k)_k \neq \text{P}$. Rules SendMessage and ReceiveMessage of $\mathcal{A}'$ are refined versions of the corresponding rules of $\mathcal{A}$. Mode S leads in both $\mathcal{A}'$ and $\mathcal{A}'$ to stuttering steps. Hence we can conclude R_k. (2) Assume $\text{mode}(\text{A}_k)_k = \text{P}$. Rules TransmitToken and InitiateProbe of $\mathcal{A}'$ change only functions from $\tau' \setminus \tau$. Rule NextProbe of $\mathcal{A}'$ changes function terminationDetected. This function is an external function of $\mathcal{A}$, i.e., can be changed by the environment in an arbitrary way. *q.e.d.*

With lemma 1 we can now conclude that ρ is a possible run of $\mathcal{A}$. It remains to show that ρ fulfills constraints C2, C3, and C4. Let t : Nat $\rightarrow$ Machine be a partial function which is defined for all time points k, denoted by $\text{Def}(\text{t}(k))$, at which a token exists and which returns for such time points the machine at which the token resides. For the sake of brevity let q_k denote tokenValue_k. The following property P is taken from [Dij99] and is defined as

$$\text{P: } \Box(\text{Def(t)} \rightarrow (\text{P0} \wedge (\text{P1} \vee \text{P2} \vee \text{P3} \vee \text{P4})))$$

where:

$$\text{P0: } B = (\sum i : 0 \leq i < N : c(i))$$

Informally, property P0 says that the sum of all local message counters is the number of all messages on their way.

$$\text{P1: } (\forall i : \text{t} < i < N : \text{active}(i) = \text{false}) \wedge (\sum i : \text{t} < i < N : c(i)) = q)$$

Call machines $i : \text{t} < i < N$ visited machines. Informally, property P1 says that all visited machines are passive and that the tokenValue is the sum of the local message counters of these machines.

P2: $(\sum i : 0 \leq i \leq \mathrm{t} : c(i)) + q > 0$

Informally, property P2 means that the sum of all local message counters of the unvisited machines plus the value of the token is greater than 0.

P3: $\exists i : 0 \leq i \leq \mathrm{t} : \mathrm{color}(i) = \mathrm{black}$

Informally, property P3 means that there exists an unvisited machine i which is black.

P4: tokenColor = black

The meaning of property P4 is clear. We write $\mathcal{A}' \models \mathrm{P}$ if each run of $\mathcal{A}'$ satisfies property P.

Lemma 2. P *is an invariant of* $\mathcal{A}'$*, i.e.,* $\mathcal{A}' \models \mathrm{P}$.

Proof. Let $\rho' = (A', S')$ be a run run of $\mathcal{A}'$. We simply say Q_k holds for a state property Q if $S'_k \models Q$ is true.
The message counter of a machine will be incremented when sending a message and decremented when receiving a message. Thus we can immediately conclude $\forall k : \mathrm{P0}_k$ or equivalently $\Box\, \mathrm{P0}$.
Let $\widehat{k}$ be an arbitrary natural number such that $t(\widehat{k})$ is defined. From the rules of $\mathcal{A}'$ follows that there must be at least one probe starting before $\widehat{k}$. More precisely, there exists a maximal $k_0 \leq \widehat{k}$ such that at time $k_0 - 1$ no machine has the token and at time k_0 machine M_{N-1} has the token.
We show:

1. the conjecture holds for k_0, and
2. if the conjecture holds for a $k < \widehat{k}$ so it holds for $k + 1$.

We start with $k = k_0$. At step $(k_0 - 1, k_0)$ the token is created. Thus M_0 executes in this move rule InitiateProbe. Hence $t(k_0) = N - 1$ and $\mathrm{P1}_{k_0}$. Assume the conjecture to be true for $k < \widehat{k}$. Thus at least one of the properties $P1_k, ..., P4_k$ holds.

(1) Assume $\mathrm{P1}_k$ holds.
  (1.1) Assume that there exists $i_0 : \mathrm{t}(k) < i_0 < N$ such that M_{i_0} receives a message in step $(k, k+1)$. Since M_{i_0} executes rule ReceiveMessage in step $(k, k+1)$ the message counter of M_{i_0} is decremented and M_{i_0} becomes active (cf. C1). Token location, token color, and token value do not change. We know that $\mathrm{c}(i_0)_{k+1} = \mathrm{c}(i_0)_k - 1$, $\mathrm{active}(i_0)_{k+1} = \mathit{true}$, and $\mathrm{t}(k) = \mathrm{t}(k+1)$ hold. Since machine i_0 receives a message we get $B_k > 0$. With $\mathrm{P1}_k$ we can conclude $(\sum i : \mathrm{t}(k+1) < i < N : \mathrm{c}(i)_{k+1}) = \mathrm{q}_k - 1$. With $\mathrm{P0}_{k+1}$ and $B_{k+1} \geq 0$ we get $\mathrm{P2}_{k+1}$.

(1.2) Assume that for all $i : \mathrm{t}(k) < i < N$ machine M_i does not receive a message in step $(k, k+1)$. At time k all machines M_i with $\mathrm{t}(k) < i < N$ are passive, thus they can not send a message and hence their message counters do not change. Their activity state does not change, too.

(1.2.1) Assume the token is not transmitted in step $(k, k+1)$. Then $\mathrm{P1}_{k+1}$ holds.

(1.2.2) Assume the token is transmitted in step $(k, k+1)$. We can conclude that rule TransmitToken is executed by machine $M_{\mathrm{t}(k)}$ and $\mathrm{t}(k) > 0$. Hence machine $M_{\mathrm{t}(k)}$ is passive both at time k and at time $k+1$ (cf. C1). The message counter of $M_{\mathrm{t}(k)}$ does not change. When $M_{\mathrm{t}(k)}$ executes rule TransmitToken it increases the token value by $\mathrm{c}(\mathrm{t}(k))$. Hence $\mathrm{P1}_{k+1}$ holds.

(2) Assume $\mathrm{P2}_k$ holds.

(2.1) Assume there exists $i_0 : 0 \leq i_0 \leq \mathrm{t}(k)$ such that M_{i_0} receives a message in step $(k, k+1)$. Then $\mathrm{color}(i_0)_{k+1} = \mathit{black}$ and $\mathrm{t}(k+1) = \mathrm{t}(k)$ holds. Thus $\mathrm{P3}_{k+1}$ holds.

(2.2) Assume for all $i : 0 \leq i \leq \mathrm{t}(k)$ M_i receives no message in step $(k, k+1)$. Thus we get $\sum i : 0 \leq i \leq \mathrm{t}(k) : \mathrm{c}(i)_{k+1} \geq \sum i : 0 \leq i \leq \mathrm{t}(k) : \mathrm{c}(i)_k$.

(2.2.1) Assume there is token transmission in step $(k, k+1)$. In this case $\mathrm{P2}_{k+1}$ holds.

(2.2.2) Assume there is no token transmission in step $(k, k+1)$. In this case we know that rule TransmitToken is executed by machine $M_{\mathrm{t}(k)}$. Furthermore we know that $\mathrm{t}(k) > 0$ holds. We get immediately $\sum i : 0 \leq i \leq \mathrm{t}(k) : \mathrm{c}(i)_{k+1} = \sum i : 0 \leq i \leq \mathrm{t}(k) : \mathrm{c}(i)_k$. Using $\mathrm{t}(k+1) = \mathrm{t}(k) - 1$ and $\mathrm{q}_{k+1} = \mathrm{q}_k + \mathrm{c}(\mathrm{t}(k))_k$ we get $(\sum i : 0 \leq i \leq \mathrm{t}(k+1) : \mathrm{c}(i)_{k+1}) + \mathrm{q}_{k+1} = (\sum : 0 \leq i \leq \mathrm{t}(k) : \mathrm{c}(i)_k) + \mathrm{q}_k$. The assumption $\mathrm{P2}_k$ gives $\mathrm{P2}_{k+1}$.

(3) Assume $\mathrm{P3}_k$ holds.

(3.1) Assume $\mathrm{t}(k+1) = \mathrm{t}(k)$ holds. We get immediately $\mathrm{P3}_{k+1}$.

(3.2) Assume $\mathrm{t}(k+1) \neq \mathrm{t}(k)$ holds.

(3.2.1) Assume $M_{\mathrm{t}(k)}$ is black at time k. The token will be blackened and transmitted to machine $M_{\mathrm{t}(k+1)}$ in step $(k, k+1)$. Thus, we get $\mathrm{P4}_{k+1}$.

(3.2.2) Assume $M_{\mathrm{t}(k)}$ is white at time k. There exists i with $0 \leq i \leq \mathrm{t}(k+1)$ such that M_i is black at time $k+1$. Thus, we get $\mathrm{P3}_{k+1}$.

(4) Assume $\mathrm{P4}_k$ holds. If Machine 0 initiates the probe it creates a new token (cf. rule InitiateProbe). This token is transmitted to machine $N-1$. The token is white and has value 0. Within a probe the token can only turn black (cf. rule TransmitToken). Thus we can conclude that $\mathrm{P4}_{k+1}$ holds.

q.e.d.

This invariant is now used to prove $\rho \models \mathrm{C2} \wedge \mathrm{C3} \wedge \mathrm{C4}$. Note that $\rho' \models \mathrm{C2} \wedge \mathrm{C3} \wedge \mathrm{C4}$ implies $\rho \models \mathrm{C2} \wedge \mathrm{C3} \wedge \mathrm{C4}$. We start with a simple lemma.

Lemma 3. $\rho \models$ C4

Proof. There exists no rule in $\mathcal{A}'$ which sets terminationDetected to false. Thus we get $\rho' \models C4$ and hence $\rho \models C4$. *q.e.d.*

Now we show that the value of terminationDetected in ρ is "correct".

Lemma 4. $\rho \models$ C2

Proof. Assume in ρ' that there exists a time point k with $\text{terminationDetected}_k =$ true. Let k_1 be the smallest number with this property. This means that at time $k_0 := k_1 - 1$ terminationDetected is false. Hence M_0 sets in step (k_0, k_1) terminationDetected to true. Thus $\text{c}(0)_{k_0} + \text{tokenValue}_{k_0} = 0$, $\text{color}(0)_{k_0} =$ white, $\text{active}(0)_{k_0} =$ false, and $\text{tokenColor}_{k_0} =$ white. With lemma 2 we know that $P1_{k_0} \vee P2_{k_0} \vee P3_{k_0} \vee P4_{k_0}$. From the above follows $\neg(P2_{k_0} \vee P3_{k_0} \vee P4_{k_0})$. Hence $P1_{k_0}$ holds. We can conclude that Termination_{k_0} holds. Since terminationDetected remains true and the termination situation is stable this leads to $\rho' \models C2$ and hence $\rho \models C2$. *q.e.d.*

Note that in each probe the token returns to machine 0 after a finite period of time if ρ' contains termination. In the following we simply say that in this case the probe ends after a finite period of time. This can be seen by constraint C5 and the rules realizing the probe, i.e., InitiateProbe, TransmitToken, and NextProbe, respectively.

Lemma 5. $\rho \models$ C3

Proof. Assume there exists in ρ' a time point k such that Termination_k holds. We know that in this case each probe ends after a finite period of time. If a probe ends at a point with no termination machine 0 initiates a new probe. Otherwise machine 0 would detect termination at a point with no termination. This is in contradiction with lemma 4. Hence lemma 4 guarantees the initiation of a new probe. Thus there exists a probe, say Pr_0, which ends within the termination.

(1) Assume M_0 detects in probe Pr_0 termination. Then we are finished.
(2) Assume M_0 detects not termination in probe Pr_0. Then a new probe Pr_1 is initiated by M_0. The token returns in probe Pr_1 with tokenValue = 0. Since upon token transmission machines whitens itself we know that all machines are white when the token returns to M_0.
  (2.1) Assume the token returns white to M_0 in Pr_1. Then M_0 detects termination.
  (2.2) Assume the token returns black to M_0 in Pr_1. Then M_0 initiates a new probe Pr_2. In this probe the token returns white to M_0 and M_0 detects termination.

q.e.d.

We can conclude that ρ is a run of $\mathcal{A}$. We have shown that each lower-level run is an implementation of a higher-level run. This subsection is summarized in the following theorem.

Theorem 1. *$\mathcal{A}'$ implements $\mathcal{A}$*

5 Conclusions

In this paper we have presented a methodology for the specification and verification of distributed algorithms using Gurevich's concept of Abstract State Machines. Starting with an informal *problem description* one constructs a *higher-level specification*, which should be an appropriate mathematical model of the problem description. The appropriateness is established by a *justification*. A *lower-level specification* represents the algorithm on a more concrete abstraction level. The *mathematical verification* guarantees that the lower-level specification implements the higher-level specification. This methodology was presented by a well-known distributed algorithm, namely the termination detection algorithm originally invented by Dijkstra, Feijen and van Gasteren in [DFvG83] in a slight variation presented in [Dij99].
In this paper we have mainly stressed on the mathematical verification. The verification is given on a detailed mathematical level. Note that the presented proofs are *not formal*, i.e., they are not based on a proof calculus. The goal of these informal proofs is to give the underlying ideas. They can be seen as abstractions from detailed and rigor formal proofs based on a proof calculus. Future research will emphasize on a *formal, mathematical verification*.
The *justification* that the higher-level specification is appropriate for the problem description is beyond the scope of this paper. Future research will emphasize on *methods for justification*. More precisely, we will investigate the FOREST-approach presented at web page [KP] (cf. [PGK97] for the underlying idea).

Acknowledgment. I would like to thank Thomas Deiß, Martin Kronenburg, and Klaus Madlener for their constructive criticism on this paper. Furthermore I would like to thank the anonymous referees for their useful and detailed comments and suggestions on the previous version of this paper.

References

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[BGR95] Egon Börger, Yuri Gurevich, and Dean Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[Bör99] Egon Börger. High level system design and analysis using abstract state machines. In Hutter, Stephan, Traverso, and Ullman, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS. Springer, 1999. to appear.

[DFvG83] Edsger W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, 1983.

[Dij99] Edsger W. Dijkstra. Shmuel Safra's version of termination detection. In M. Broy and R. Steinbrüggen, editors, *Proceedings of the NATO Advanced Study Institute on Calculational System Design, Marktoberdorf, Germany, 28 July - 9 August 1998*, pages 297–301, 1999.

[DS80] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur97] Yuri Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, 1997.

[Gur99] Yuri Gurevich. The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science*, 67:93–124, February 1999. Columns: Logic in Computer Science.

[KP] Martin Kronenburg and Christian Peper. The FOREST Approach: World Wide Web page at `http://rn.informatik.uni-kl.de/~forest/`.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[PGK97] Christian Peper, Reinhard Gotzhein, and Martin Kronenburg. A generic approach to the formal specification of requirements. In *1st IEEE International Conference on Formal Engineering Methods 1997 (ICFEM'97), Hiroshima, Japan*. IEEE Computer Society, 1997.