On Engineering a Distributed Algorithm

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
http://www.risc.uni-linz.ac.at/people/schreine

August 20, 1998

Abstract

The aim of this paper to demonstrate rigorous reasoning in the context of concurrency. We present a small case study on the engineering of a distributed algorithm: its derivation from the problem specification, formal specifications of its behavior and of the desired correctness property, and rigorous proofs that the behavior always meets this property and eventually leads to a desired state.

Keywords: distributed algorithms, concurrency, temporal logic, specification, verification.

Contents

1 Introduction

We present a small case study on the engineering of a distributed algorithm: its derivation from the problem specification, formal specifications of its behavior and of the desired correctness property, and rigorous proofs that the behavior always meets this property and eventually leads to a desired state.

This paper was inspired by a lecture of E.W. Dijkstra and corresponding handouts which derived in 45 minutes respectively on eight hand-written A5 pages a distributed algorithm from the problem specification [Dij98]. The derivation starts with a simple algorithmic idea and a corresponding invariant that captures the state of the system at any time and that implies the desired property at termination. However, the initial algorithm cannot maintain the stated invariant; both the algorithm and the invariant are therefore gradually refined (always implying correctness at termination) until algorithm and invariant match.

This paper puts the high-level reasoning shown in that lecture into a formal framework in which we are able to specify every detail of the algorithm and to rigorously reason about its correctness. The chosen language is essentially (i.e., in spirit, not in all syntax) the Temporal Logic of Actions [Lam94] which specifies a concurrent system by a logic formula that describes every possible behavior of the system. Reasoning about the system is ultimately reduced to reasoning in classical first-order predicate logic.

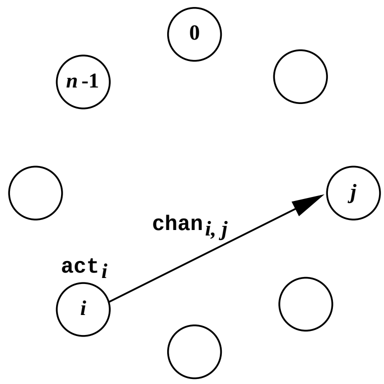
At first glance, it may sound strange to call such a process "engineering" since this term is in computer science usually associated to heuristics, practices, and supporting tools that experts consider useful for constructing large software systems. Since the underlying algorithms are often rather simple and intuitively well understood, the algorithmic side of the problem tends to be neglected and rigorous reasoning is rarely applied.

Likewise sometimes the impression exists that the central problem of engineering distributed software systems is that of developing suitable programming languages, monitoring and debugging tools, standardized communication protocols, component interfaces, and the like. While these are undoubtedly important aspects, this view captures only one side, that of *technology*. Another at least equally important side is that of *thinking* in the context of distributed systems, which is difficult because one has to consider concurrent activities that may interfere with each other. Reasoning by "hand-waving" is here dangerous and ultimately yields erroneous and unreliable software.

In order to build industrial-strength reliable distributed systems [Bir96], a software engineer needs a mental skeleton that helps her to cope with the issues related to concurrency, a framework within she can express and refine her ideas, and rules that allow heer to reason about their correctness. Even if the resulting algorithm is actually never formally verified in detail, this skeleton helps one to focus on the important aspects of the design and allows one to increase confidence in its correctness by checking the most critical parts. An integrated part of the engineering of distributed systems is therefore the modeling of concurrent activities and the reasoning about them.

It is the aim of this paper to demonstrate rigorous reasoning in the context of a distributed system. For this purpose, we use a formal language that give a

Figure 1: The Basic System



unique meaning to concepts that can be only ambiguously expressed in natural language (which we however extensively use for motivation and comments). Proofs are presented in a semi-formal style suitable for human reading; some go into large detail, while some contain just the crucial information that should enable anyone to fill in the missing parts. Chapter 2 introduces the formal language that we will use to specify the problem and the algorithm; Chapter 3 is essentially a transcription of Dijkstra's derivation in this language; Chapter 4 presents a proof of the partial correctness of the algorithm; Chapter 5 includes a corresponding termination proof.

2 The System Model

Problem Statement We are given a system with n processes numbered $\mathtt{0}$ to $n{-}1$. Each process can send via buffered communication channels messages to all other processes, i.e., there exist n^2 channels between all pairs of processes[1]. Initially all processes are *active*; a process may send messages only, if it is active. An active process may spontaneously decide to become inactive (but not vice versa); on receipt of a message, a process becomes active again.

Our problem is to design an algorithm that allows a process, say process $\mathtt{0}$, to detect whether the system has *terminated*, i.e., whether all processes in the system are inactive and all channels are empty. If the system has terminated, no process will ever become active again.

We model the system with the help of the following variables (see Figure 1):

$$\mathtt{act}{\in}\mathbb{Z}_n{\to}\mathbb{B},$$
$$\mathtt{chan}{\in}\mathbb{Z}_n{\times}\mathbb{Z}_n{\to}\mathbf{Seq}(MSG).$$

The variable $\mathtt{act}$ represents a predicate (a boolean-valued function) such that $\mathtt{act}_i$ is true if and only if process i is active ($\mathbb{Z}_n$ denotes the set $\{\mathtt{0}, \ldots, n{-}1\}$).

[1] For simplicity, we allow a process to send messages to itself. The results in this paper are not affected if we restrict communication to pairs of different processes.

The variable $\mathtt{chan}$ represents the system channels, i.e., $\mathtt{chan}_{i,j}$ holds a sequence of messages $\langle m_0, \ldots m_{l-1}\rangle$ (where every m_i is an element of some domain MSG) that were sent by process i but have not yet been received by process j.

Temporal Logic For specifying the system and the termination detection algorithm, we use the language of *temporal logic* in a style close to TLA [Lam94]. A system is identified with the set of its possible *behaviors* where a behavior σ is an infinite sequence of states

$$\sigma = \langle \sigma_0, \sigma_1, \ldots \rangle$$

such that each σ_i describes the values of all program variables in step i. For instance,

$$\sigma_i(\mathtt{act}_k) = \mathsf{false}$$

says that in step i process k is inactive.

We describe the set of system behaviors by logical formula of the following structure

$$\begin{array}{l} \wedge\ Init_n(\mathtt{act}, \mathtt{chan}) \\ \wedge\ \Box[Action_n(\mathtt{act}, \mathtt{chan})] \end{array}$$

where a list of predicates bulleted with $\wedge$ denotes the conjunction of these predicates (and analogously for disjunction). The rest of this section explains the construction of this formula.

Initial State Condition The *initial state condition Init* describes the values of the variables $\mathtt{act}$ and $\mathtt{chan}$ in state σ_0, i.e.,

$$\begin{array}{l} Init_n(\mathtt{act}, \mathtt{chan}) :\Leftrightarrow \\ \wedge\ \forall i{\in}\mathbb{Z}_n\colon \mathtt{act}_i \\ \wedge\ \forall i{\in}\mathbb{Z}_n,\ j{\in}\mathbb{Z}_n\colon \mathtt{chan}_{i,j} = \langle\rangle. \end{array}$$

Since *Init* only constrains the variables in a single state, it is a so called *state predicate.*

Actions The predicate *Action* models the transition from one state to the next, i.e., it describes a relation between any two subsequent states σ_i and σ_{i+1}. We call such a predicate an *action.* An action is denoted by a formula like

$$\begin{array}{l} \wedge\ \mathtt{x}' = \mathtt{x}{+}\mathtt{y} \\ \wedge\ \mathtt{y}' = \mathtt{y}{+}1 \end{array}$$

which describes the relationship between the values of the variables $\mathtt{x}$ and $\mathtt{y}$ in state σ_i and the values of these variables in state σ_{i+1} (denoted by $\mathtt{x}'$ and $\mathtt{y}'$). Above formula therefore means

$$\wedge\ \sigma_{i+1}(x) = \sigma_i(\mathtt{x}) + \sigma_i(\mathtt{y})$$
$$\wedge\ \sigma_{i+1}(y) = \sigma_i(\mathtt{y}) + 1$$

and roughly corresponds to the programming language statement

```
x = x+y;
y = y+1;
```

An action A is *enabled* in some state, written as *Enabled* A, if there exists a possible successor state such that the action models the relationship between both states (i.e., if an action is not enabled in a state, it is not possible to execute the action in that state). For instance, the action

$$\mathtt{c} = \langle \mathtt{m}' \rangle \circ \mathtt{c}'$$

is only enabled in a state in which `c` holds a non-empty sequence c; in the successor state `m` holds the first element of c and `c` holds the rest of c.

We denote by

$$f[i \mapsto v]$$

the function g that is identical to f except for $g_i = v$. The action

$$\mathtt{a} = \mathtt{a}[i \mapsto \mathtt{a}_i + 1]$$

thus roughly corresponds to the programming language statement

```
a[i] = a[i]+1;
```

Temporal Formulas The temporal formula $\Box P$ ("always P") describes a property of whole behaviors: if P is a state predicate, then P is true for every state σ_i in every behavior σ; if P is an action, then P is true for every pair of states σ_i and σ_{i+1}; iff P is itself a temporal formula, P is true for every suffix $\langle \sigma_i, \sigma_{i+1}, \ldots \rangle$ of the behavior.

A formula of the form $\Box P$ turns out to be too restrictive to specify concurrent systems: we want to have $S \Rightarrow T$ whenever system S "implements" or "refines" system T, i.e., S must fulfill all constraints of T but may e.g. introduce new variables and actions on these variables. However, then a step of S may only change variables that are not visible in T, which is not allowed by the implication if T is specified by a formula $\Box P$ that constrains *every* step of the system.

We therefore introduce the operator $[P]$ informally defined as "if not P, then all visible variables remain unchanged". The "next-state" relation of a system is then specified by a formula $\Box[P]$, which allows "stuttering" steps, i.e., sequences of states in which none of the visible variables changes (see below on the definition of "visibility").

The temporal formula $\Diamond P$ ("eventually P") is a shortcut for $\neg\Box\neg P$, i.e., P must not be false forever. The operator $\langle P \rangle$ is informally defined as "P and some of the visible variables changes".

Specification and Action Parameters The operators [.] and $\langle . \rangle$ depend on the set of program variables that are "visible" in a specification. In order to make this set explicit, we parameterize every system specification by the set of all visible variables and interpret each occurence of these operators with respect to this set. For instance,

$$\mathit{System}_{n\in\mathbb{N}}(\mathtt{act}\in\mathbb{Z}_n\to\mathbb{B},\ \mathtt{chan}\in\mathbb{Z}_n\times\mathbb{Z}_n\to\mathbf{Seq}(\mathit{MSG}))$$

has `act` and `chan` as visible program variables. As a visual and mental aid, we usually include the expected variable types in such parameter specifications. We use subscripts to denote parameterization by mathematical variables, such as the mathematical variable n in above specification.

Likewise, each action defined in a specification is parameterized by the variables referenced in this action. For instance, the action *Inactivate* defined in

$\mathit{System}_{n\in\mathbb{N}}(\mathtt{act}\in\mathbb{Z}_n\to\mathbb{B},\ \mathtt{chan}\in\mathbb{Z}_n\times\mathbb{Z}_n\to\mathbf{Seq}(\mathit{MSG}))$:⇔
  let
    $\mathit{Inactivate}_i(\mathtt{act})$:⇔
      $\mathtt{act}' = \mathtt{act}[i \mapsto \mathsf{false}]$
    ...
  ...

only refers to the variable `act`. The body of the action is thus implicitly conjoined with conditions that state that all variables remain unchanged that are visible in the specification but are not referenced in the body of the action, i.e.,

$$\mathtt{chan}' = \mathtt{chan}$$

in above case.

We usually also tag every parameter of an action by a "qualifier":

- **in** p signals that the variable p occurs in the corresponding formula only in the unprimed form, i.e., the variable only appears as part of the enabling condition of the action constraining its value only in the "current" state.
- **out** p signals that p occurs in the formula only in the primed form, i.e., it constrains the value of the variables only the "successsor" state.
- **io** p signals that p occurs in the formula both in its primed and in its unprimed form and thus describes a relationship between the values of the variable in both states.

Apparently a state predicate (such as the initial condition) only has **in** qualifiers. We freely skip the qualifier when we do not consider it useful in some context.

Specification 1: The Basic System

$System_{n\in\mathbb{N}}(\mathtt{act}\in\mathbb{Z}_n\to\mathbb{B},\ \mathtt{chan}\in\mathbb{Z}_n\times\mathbb{Z}_n\to\mathbf{Seq}(MSG))$:⇔
(a system with n processes, active or inactive, connected by channels)
let

$Init(\mathbf{in}\ \mathtt{act},\ \mathbf{in}\ \mathtt{chan})$:⇔
(all processes are active, all channels are empty)
$\wedge\ \forall i\in\mathbb{Z}_n\colon \mathtt{act}_i$
$\wedge\ \forall i\in\mathbb{Z}_n,\ j\in\mathbb{Z}_n\colon \mathtt{chan}_{i,j} = \langle\rangle$,
$Send_{i,j}(\mathbf{in}\ \mathtt{a},\ \mathbf{io}\ \mathtt{chan})$:⇔
(active process i sends some message to process j)
$\wedge\ \mathtt{a}$
$\wedge\ \exists m\in MSG\colon \mathtt{chan}' = \mathtt{chan}[(i,j)\mapsto \mathtt{chan}_{i,j}\circ\langle m\rangle]$,
$Receive_{i,j}(\mathbf{io}\ \mathtt{act},\ \mathbf{io}\ \mathtt{chan})$:⇔
(having received some message from process i, process j is active)
$\wedge\ \mathtt{act}' = \mathtt{act}[j\mapsto \mathsf{true}]$
$\wedge\ \exists m\in MSG\colon \mathtt{chan} = \mathtt{chan}'[(i,j)\mapsto \langle m\rangle\circ\mathtt{chan'}_{i,j}]$,
$Inactivate_i(\mathbf{io}\ \mathtt{act})$:⇔
(process i becomes inactive)
$\mathtt{act}' = \mathtt{act}[i\mapsto \mathsf{false}]$,
$Action_{n,i}(\mathbf{io}\ \mathtt{act},\ \mathbf{io}\ \mathtt{chan})$:⇔
(process i is involved in some action)
$\vee\ \exists j\in\mathbb{Z}_n\colon Send_{i,j}(\mathbf{in}\ \mathtt{act}_i,\ \mathbf{io}\ \mathtt{chan})$
$\vee\ \exists j\in\mathbb{Z}_n\colon Receive_{i,j}(\mathbf{io}\ \mathtt{act},\ \mathbf{io}\ \mathtt{chan})$
$\vee\ Inactivate_i(\mathbf{io}\ \mathtt{act})$:
(initial state condition and transition relation)
$\wedge\ Init(\mathbf{in}\ \mathtt{act},\ \mathbf{in}\ \mathtt{chan})$
$\wedge\ \Box[\exists i\in\mathbb{Z}_n\colon Action_{n,i}(\mathbf{io}\ \mathtt{act},\ \mathbf{io}\ \mathtt{chan})]$

The System Specification Having introduced the language of our specifications, we now continue with defining the actions of the system sketched at the start of this section:

$Inactivate_i(\mathbf{io}\ \mathtt{act})$:⇔
$\mathtt{act}' = \mathtt{act}[i\mapsto \mathsf{false}]$,

says that process i may become inactive at any time (if it is already inactive, this action does not have any effect). Sending a message is expressed by

$Send_{i,j}(\mathbf{in}\ \mathtt{a},\ \mathbf{io}\ \mathtt{chan})$:⇔
$\wedge\ \mathtt{a}$
$\wedge\ \exists m\in MSG\colon \mathtt{chan}' = \mathtt{chan}[(i,j)\mapsto \mathtt{chan}_{i,j}\circ\langle m\rangle]$,

which is only enabled in a state where $\mathtt{a}$ (to be instantiated by $\mathtt{act}_i$) holds. In the successor state, some message m is appended to the current content of channel $\mathtt{chan}_{i,j}$. The action

$$\begin{array}{l} \mathit{Receive}_{i,j}(\textbf{io}\ \mathtt{act},\ \textbf{io}\ \mathtt{chan}) :\Leftrightarrow \\ \quad \wedge\ \mathtt{act}' = \mathtt{act}[j \mapsto \mathsf{true}] \\ \quad \wedge\ \exists m{\in}MSG\colon\ \mathtt{chan} = \mathtt{chan}'[(i,\, j) \mapsto \langle m\rangle{\circ}\mathtt{chan}'_{i,j}] \end{array}$$

is only enabled in a state where $\mathtt{chan}_{i,j}$ holds a message m. In the successor state, m is removed from this channel and $\mathtt{act}_j$ is set to true (i.e., an inactive process becomes active again).

The complete system description is compiled in Specification 1.

3 The Termination Detection Algorithm

We will extend the *System* interface to

$$\mathit{SystemT}_{n\in\mathbb{N}}(\mathtt{act}{\in}\mathbb{Z}_n{\to}\mathbb{B},\ \mathtt{chan}{\in}\mathbb{Z}_n{\times}\mathbb{Z}_n{\to}\textbf{Seq}(MSG),\ \mathtt{term}{\in}\mathbb{B})$$

such that $\mathtt{term}$ initialized by

$$\begin{array}{l} \mathit{Init}(\ldots) :\Leftrightarrow \\ \quad \wedge\ \ldots \\ \quad \wedge\ \neg\mathtt{term} \end{array}$$

signals termination of the system.

Basic Idea The derivation of the termination detection algorithm is based on the following idea (see Figure 2): If Process $\mathsf{0}$ wants to detect termination, it sends a signal to process $n{-}1$. If an inactive process $i > \mathsf{0}$ receives the signal, it forwards the signal to process $i{-}1$. An active process keeps the signal until it becomes inactive. The signal therefore represents a "token" that circulates through the ring of processes.

We model this circulation by introducing a variable

$$\mathtt{sig}{\in}\mathbb{Z}_n{\to}\mathbb{B}$$

initialized by

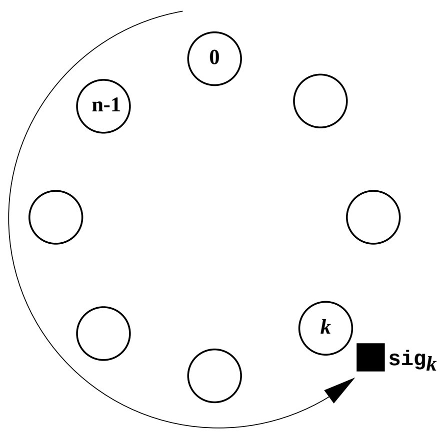
$$\begin{array}{l} \mathit{Init}(\ldots) :\Leftrightarrow \\ \quad \wedge\ \ldots \\ \quad \wedge\ \forall i{\in}\mathbb{Z}_n\colon\ \neg\mathtt{sig}_i \end{array}$$

and by two actions

$$\begin{array}{l} \mathit{Start}_i(\ldots) :\Leftrightarrow \\ \quad \wedge\ i = \mathsf{0} \\ \quad \wedge\ \mathtt{sig}' = \mathtt{sig}[i{\ominus}_n 1 \mapsto \mathsf{true}] \end{array}$$

$$\mathit{Forward}_i(\ldots) :\Leftrightarrow$$

Figure 2: Token Circulation



$\wedge$ $\mathtt{sig}' = \textbf{if } i \neq 0 \textbf{ then } \mathtt{sig}[i \mapsto \mathsf{false}, i \ominus_n 1 \mapsto \mathsf{true}] \textbf{ else } \mathtt{sig}[i \mapsto \mathsf{false}]$

where $a \ominus_n b$ denotes the difference modulo n, i.e., $0 \ominus_n 1 = n-1$).

(Modeling token circulation by an array of boolean signals reflects the behavior of the distributed algorithm closer than modeling it by a single integer position that "automatically" guarantees token unicity.)

However, above specification allows Process 0 to submit a new token before it has received the token it has previously submitted. We therefore introduce another variable

$\mathtt{run} \in \mathbb{B}$

that records whether there is a token in the system and then disables *Start*:

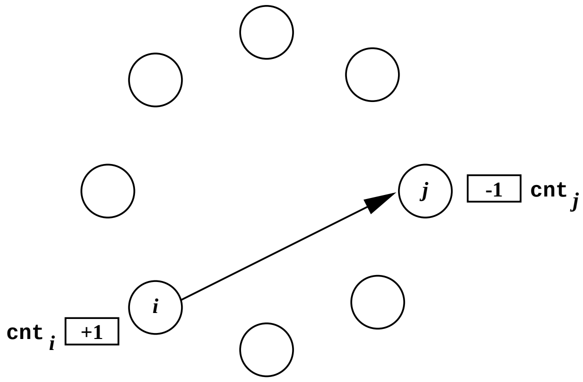
$Start_i(\dots) :\Leftrightarrow$
$\wedge \dots$
$\wedge \neg\mathtt{run} \wedge \mathtt{run}'$
$Forward_i(\dots) :\Leftrightarrow$
$\wedge \dots$
$\wedge \mathtt{run}' = \textbf{if } i \neq 0 \textbf{ then } \mathtt{run} \textbf{ else } \mathsf{false}$

However, when Process 0 finally receives the token, it can only deduce that each process has been inactive at some time in the past. Since a process mave have received a message after forwarding the token, it may have become active again. Only if the algorithm could maintain the invariant

$\forall k \in \mathbb{Z}_n \colon \mathtt{sig}_k \Rightarrow$
$\forall j \in \mathbb{Z}_n \colon j > k \Rightarrow \neg\mathtt{act}_j,$

Process 0 could deduce from the receipt of the token (i.e., $\mathtt{sig}_0$) and its own inactivity (i.e., $\neg\mathtt{act}_0$) that all processes are currently inactive.

Figure 3: Message Counters



Message Counters Even if Process 0 receives the token under above invariant, there may be still messages pending in the network that can cause processes to become active again. We therefore need some mean to keep track of the number of messages pending in the network. Since a process only knows about the messages this it itself sends or receives, this can be only achieved by introducing a distributed counter

$$\mathtt{cnt} \in \mathbb{Z}_n \rightarrow \mathbb{Z}$$

such that every process sending a message increases its counter and every process receiving a message decreases its counter (see Figure 3):

$$\begin{aligned}
&\mathit{Send}_{i,j}(\ldots) \;:\Leftrightarrow\\
&\quad\wedge\ \ldots\\
&\quad\wedge\ \mathtt{cnt}' = \mathtt{cnt}[i \mapsto \mathtt{cnt}_i{+}1],\\
&\mathit{Receive}_{i,j}(\ldots) \;:\Leftrightarrow\\
&\quad\wedge\ \ldots\\
&\quad\wedge\ \mathtt{cnt}' = \mathtt{cnt}[j \mapsto \mathtt{cnt}_j{-}1]
\end{aligned}$$

Then clearly the total sum of the distributed counter equals the number of the messages in the network, i.e., the system maintains the invariant

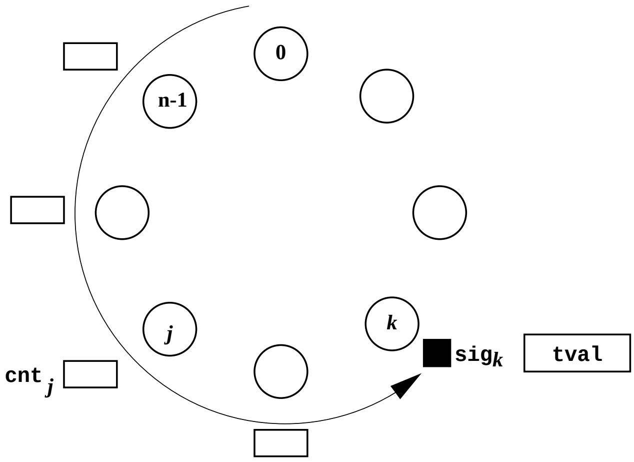
$$\textstyle\sum_{i \in \mathbb{Z}_n} \mathtt{cnt}_i = \sum_{i \in \mathbb{Z}_n} \sum_{j \in \mathbb{Z}_n} \mathit{len}(\mathtt{chan}_{i,j})$$

where $\mathit{len}(\mathtt{ch}) := \textbf{such } l \in \mathbb{N}\colon \exists m_0, \ldots, m_{l-1}\colon \mathtt{ch} = \langle m_0, \ldots, m_{l-1} \rangle$.

Token Value How can Process 0 learn about this sum? Apparently this is only possible, if the circulating token collects this information and delivers it to this process. We therefore introduce

$$\mathtt{tval} \in \mathbb{Z}$$

Figure 4: Token Value



modeling the value carried by the token. Process 0 initializes the value to 0 when submitting the token; each forwarding process adds the value of its counter (see Figure 4):

$$
\begin{aligned}
&\mathit{Start}_i(\ldots) :\Leftrightarrow \\
&\quad \wedge \ \ldots \\
&\quad \wedge \ \mathtt{tval}' = 0 \\
&\mathit{Forward}_i(\ldots) :\Leftrightarrow \\
&\quad \wedge \ \ldots \\
&\quad \wedge \ \mathtt{tval}' = \mathtt{tval}+\mathtt{cnt}_i
\end{aligned}
$$

If the algorithm could maintain the invariant

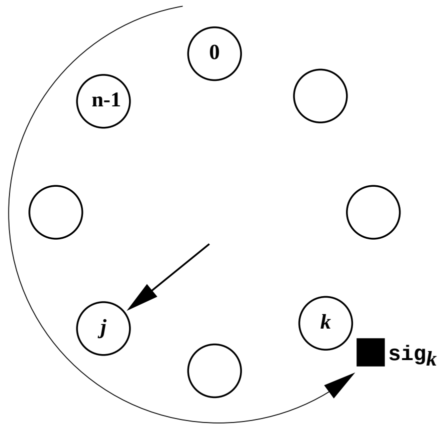
$$
\begin{aligned}
&\forall k \in \mathbb{Z}_n\colon \mathtt{sig}_k \Rightarrow \\
&\quad \wedge \ \forall j \in \mathbb{Z}_n\colon j > k \Rightarrow \neg\mathtt{act}_j \\
&\quad \wedge \ \mathtt{tval} = \textstyle\sum_{j \in \mathbb{Z}_n, j > k} \mathtt{cnt}_j
\end{aligned}
$$

then Process 0 could on receipt of the token deduce from $\neg\mathtt{act}_0$ that all processes are inactive and from $\mathtt{cnt}_0+\mathtt{tval} = 0$ that no message is in the network, i.e., that the system has terminated.

Unfortunately, the algorithm cannot maintain this invariant. We therefore have to disjoin the conclusion part (the "core") of the invariant with some weakening conditions such that the algorithm is able to maintain the weaker invariant but Process 0 can still conclude termination from the information stored locally and received by the token.

Awaking a Process Let us investigate how above invariant can be falsified by an action (i.e., the invariant holds in the state before the action but does not any more hold in the state after the action). Since all processes that have already forwarded the token are inactive, the only possibility is that such a process

Figure 5: Invalidating the Core Invariant



receives a message and becomes active again (see Figure 5). This however means that there must be still pending messages in the network. Since the invariant still holds before invalidation, a weaker version of the variant is

$$
\begin{array}{l}
\forall k \in \mathbb{Z}_n\colon \texttt{sig}_k \Rightarrow \\
\quad \vee\ \wedge\ \forall j \in \mathbb{Z}_n\colon j > k \Rightarrow \neg\texttt{act}_j \\
\quad\quad \wedge\ \texttt{tval} = \sum_{j \in \mathbb{Z}_n, j > k} \texttt{cnt}_j \\
\quad \vee\ 0 < \texttt{tval} + \sum_{j \in \mathbb{Z}_n, j \leq k} \texttt{cnt}_j
\end{array}
$$

If Process 0 receives the token (i.e., $\texttt{sig}_0$ holds), it can still conclude termination from $\neg\texttt{act}_0$ and $\texttt{cnt}_0 + \texttt{tval} = 0$.

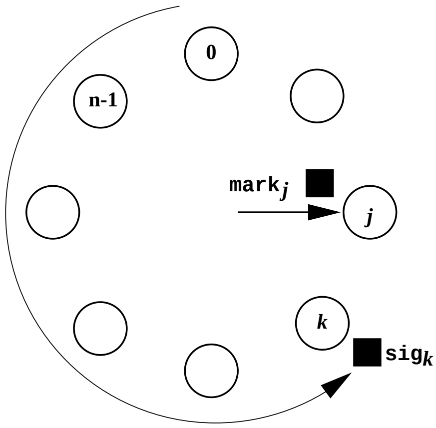
Process Marks Is it still possible for an action to falsify the weaker invariant? Clearly it can be invalidated, if a process that has not yet forwarded the token receives a message, which causes the summation term in above condition to drop by one (see Figure 6). A simple way to record whether such an incident has taken place is to associate to each process a marker that is set on message receipt. We therefore introduce

$$\texttt{mark} \in \mathbb{Z}_n \rightarrow \mathbb{B}$$

which is initialized to false in every position and which is set to true when a message is received:

$$
\begin{array}{l}
\mathit{Init}(\ldots)\ :\Leftrightarrow \\
\quad \wedge\ \ldots \\
\quad \wedge\ \forall i \in \mathbb{Z}_n\colon \neg\texttt{mark}_i \\
\mathit{Receive}_{i,j}(\ldots)\ :\Leftrightarrow \\
\quad \wedge\ \ldots \\
\quad \wedge\ \texttt{mark}' = \texttt{mark}[j \mapsto \mathsf{true}],
\end{array}
$$

Figure 6: Process Markers



Then we can weaken the invariant further to

$$
\begin{array}{l}
\forall k \in \mathbb{Z}_n\colon \mathtt{sig}_k \Rightarrow \\
\quad \vee\ \wedge\ \forall j \in \mathbb{Z}_n\colon j > k \Rightarrow \neg \mathtt{act}_j \\
\qquad \wedge\ \mathtt{tval} = \sum_{j \in \mathbb{Z}_n, j > k} \mathtt{cnt}_j \\
\quad \vee\ 0 < \mathtt{tval} + \sum_{j \in \mathbb{Z}_n, j \leq k} \mathtt{cnt}_j \\
\quad \vee\ \exists j \in \mathbb{Z}_n\colon j \leq k \wedge \mathtt{mark}_j
\end{array}
$$

If Process $\mathtt{0}$ receives the token, i.e., $\mathtt{sig_0}$ holds, it can still conclude termination from $\neg\mathtt{act_0}$, $\mathtt{cnt_0}+\mathtt{tval} = 0$, and $\neg\mathtt{mark_0}$.

Token Marker Clearly this weakened invariant is still falsified when the marked process with the least index forwards the token (see Figure 7). Therefore the token itself has to record the information when it passes a marked process. We introduce a token marker

$$\mathtt{tmark} \in \mathbb{B}$$

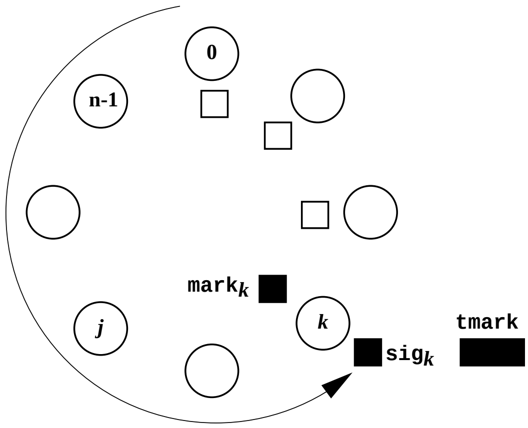
which is initialized by Process $\mathtt{0}$ and forwarded by each process as

$$
\begin{array}{l}
\mathit{Start}_i(\ldots) :\Leftrightarrow \\
\quad \wedge\ \ldots \\
\quad \wedge\ \neg\mathtt{tmark}' \\
\mathit{Forward}_i(\ldots) :\Leftrightarrow \\
\quad \wedge\ \ldots \\
\quad \wedge\ \mathtt{tmark}' = \mathtt{tmark} \vee \mathtt{mark}_i
\end{array}
$$

Then we can weaken the invariant further by

$$\forall k \in \mathbb{Z}_n\colon \mathtt{sig}_k \Rightarrow$$

Figure 7: Token Marker



$$
\begin{aligned}
&\vee \ \wedge \ \forall j{\in}\mathbb{Z}_n\colon\ j > k \Rightarrow \neg\texttt{act}_j \\
&\quad\ \wedge \ \texttt{tval} = \textstyle\sum_{j\in\mathbb{Z}_n, j>k}\texttt{cnt}_j \\
&\vee \ 0 < \texttt{tval}{+}\textstyle\sum_{j\in\mathbb{Z}_n, j\le k}\texttt{cnt}_j \\
&\vee \ \exists j{\in}\mathbb{Z}_n\colon\ j \le k \wedge \texttt{mark}_j \\
&\vee \ \texttt{tmark}
\end{aligned}
$$

This invariant cannot be falsified any more by the activity of any process.

Process 0 can then conclude termination from $\texttt{sig}_0$, $\neg\texttt{act}_0$, $\texttt{cnt}_0{+}\texttt{tval} = 0$, $\texttt{cnt}_0{+}\texttt{tval} = 0$, and $\neg\texttt{tmark}$. By definition of $\texttt{tmark}'$ and $\texttt{tval}'$, this can be stated as

$$
\begin{aligned}
&\mathit{Forward}_i(\ldots) :\Leftrightarrow \\
&\quad \wedge \ \ldots \\
&\quad \wedge \ \texttt{term}' \Leftrightarrow \textbf{if } i \neq 0 \textbf{ then } \texttt{term} \textbf{ else } (\neg\texttt{tmark}' \wedge \texttt{tval}' = 0).
\end{aligned}
$$

Resetting Process Markers If Process 0 cannot yet conclude termination after a termination detection round, it may later start another round. Without resetting the process markers, however, such an attempt is doomed to fail. Fortunately, after it has marked the token, the process marker has fulfilled its task and can be reset:

$$
\begin{aligned}
&\mathit{Forward}_i(\ldots) :\Leftrightarrow \\
&\quad \wedge \ \ldots \\
&\quad \wedge \ \texttt{mark}' = \texttt{mark}[i \mapsto \mathsf{false}]
\end{aligned}
$$

Since this action takes place simultaneously with token transmission (i.e., $\neg\texttt{sig}'_i$ holds), it cannot falsify the invariant.

If the system terminates during a termination detection round, Process 0 may not yet conclude termination at the end of this round. However, after this run, no process is active, there are no messages in the network any more, and the sum of the message counters equals `0`. Still some processes may be marked; the next termination detection round may thus fail, too. Anyway, after this run all processes are unmarked, and the next termination detection round will succeed.

Algorithm Specification The complete algorithm is compiled in Specification 2. The existential quantifier

var `v`$\in T$: ...

introduces a local program variable `v` in a formula like $(\exists v \in T: \ldots)$ introduces a local mathematical variable v. The crucial difference between both kinds is that the "variable" `v` may have different values in different states of a program behavior while the "constant" v has always the same value in all states.

4 Correctness of the Algorithm

Basics The derivation of the termination detection algorithm by repeated refinement of its invariant gives good reason to believe that the algorithm indeed solves the stated problem. However, the development was based on "high-level" reasoning about a concurrent system with rather large inference steps. We cannot be really sure that the algorithm is correct unless we formalize our reasoning, i.e., create a map for our thoughts, and decompose the large inference steps into smaller ones whose correctness is much more self-evident. A formal verification of a concurrent algorithm is tedious, because at every step some process may perform an action that violates the invariant. We have to convince ourselves that this is not the case. During this process we get a much deeper insight into the algorithm and its behavior because many subtleties are exposed that may be overlooked in a high-level presentation.

The core property to be proved is defined by a formula of the form $\Box[P]$. The basic strategy to prove such a property is based on induction: we prove that

1. the initial state condition implies P,
2. if P holds and the system performs an action, then P' holds (where P' is derived from P by replacing every program variable by its primed counterpart).

However, even if P is invariant, one may not be able to prove this fact, because P on its own may be to weak to show the second part of above rule, i.e., because P is not *inductive*. The solution is to find a stronger property I such that I is inductive and thus $\Box[I]$ can be proven. Since $I \Rightarrow P$, we can then conclude $\Box[P]$.

The invariant constructed in the previous section is an attempt to such an inductive property but it will turn out that I alone is also not sufficient. Strengthening I to some inductive I' will however make the proof quite clumsy. Instead we will repeatedly apply the same principle that we use in the proof of P: we construct

Specification 2: The Termination Detection Algorithm

$\mathit{SystemT}_{n\in\mathbb{N}}$(act$\in\mathbb{Z}_n\to\mathbb{B}$, chan$\in\mathbb{Z}_n\times\mathbb{Z}_n\to$**Seq**($\mathit{MSG}$), term$\in\mathbb{B}$) :$\Leftrightarrow$
(a system that may detect its termination)
let
Init(**in** act, **in** chan, **in** term, **in** run, **in** cnt, **in** sig, **in** mark) :$\Leftrightarrow$
$\wedge$ ... *(as in the basic system)*
$\wedge$ $\neg$term $\wedge$ $\neg$run
$\wedge$ $\forall i\in\mathbb{Z}_n$: $\mathtt{cnt}_i = 0 \wedge \neg\mathtt{sig}_i \wedge \neg\mathtt{mark}_i$,
$\mathit{Send}_{i,j}$(**in** a, **io** chan, **io** cnt) :$\Leftrightarrow$
$\wedge$ ... *(as in the basic system)*
$\wedge$ $\mathtt{cnt}' = \mathtt{cnt}[i \mapsto \mathtt{cnt}_i{+}1]$,
$\mathit{Receive}_{i,j}$(**io** act, **io** chan, **io** cnt, **io** mark) :$\Leftrightarrow$
$\wedge$ ... *(as in the basic system)*
$\wedge$ $\mathtt{cnt}' = \mathtt{cnt}[j \mapsto \mathtt{cnt}_j{-}1]$
$\wedge$ $\mathtt{mark}' = \mathtt{mark}[j \mapsto \mathsf{true}]$,
$\mathit{Inactivate}_i$(**io** act) :$\Leftrightarrow$ ... *(as in the basic system)*
Start_i(**io** run, **io** sig, **io** mark, **io** tval, **out** tmark) :$\Leftrightarrow$
(Process 0 *starts termination detection)*
$\wedge$ $i = 0 \wedge \neg\mathtt{run}$
$\wedge$ $\mathtt{run}'$
$\wedge$ $\mathtt{sig}' = \mathtt{sig}[i\ominus_n 1 \mapsto \mathsf{true}]$
$\wedge$ $\mathtt{mark}' = \mathtt{mark}[i \mapsto \mathsf{false}]$
$\wedge$ $\mathtt{tval}' = 0$
$\wedge$ $\neg\mathtt{tmark}'$,
$\mathit{Forward}_i$(**in** a, **in** c, **io** sig, **io** mark, **io** tval, **io** tmark, **io** term) :$\Leftrightarrow$
(process i gets token, forwards it, or detects termination)
$\wedge$ $\neg\mathtt{a} \wedge \mathtt{sig}_i$
$\wedge$ $\mathtt{run}' =$ **if** $i \neq 0$ **then** run **else** false
$\wedge$ $\mathtt{sig}' =$ **if** $i \neq 0$ **then** $\mathtt{sig}[i \mapsto \mathsf{false},\ i\ominus_n 1 \mapsto \mathsf{true}]$ **else** $\mathtt{sig}[i \mapsto \mathsf{false}]$
$\wedge$ $\mathtt{mark}' = \mathtt{mark}[i \mapsto \mathsf{false}]$
$\wedge$ $\mathtt{tval}' = \mathtt{tval}{+}\mathtt{c}$
$\wedge$ $\mathtt{tmark}' = \mathtt{tmark} \vee \mathtt{mark}_i$
$\wedge$ $\mathtt{term}' \Leftrightarrow$ **if** $i \neq 0$ **then** term **else** $(\neg\mathtt{tmark}' \wedge \mathtt{tval}' = 0)$,
$\mathit{Action}_{n,i}$(**io** act, **io** chan, **io** term, **io** run, **io** cnt, **io** sig, **io** mark, **io** tval, **io** tmark) :$\Leftrightarrow$
$\vee$ $\exists j\in\mathbb{Z}_n$: $\mathit{Send}_{i,j}$(**in** $\mathtt{act}_i$, **io** chan, **io** cnt)
$\vee$ $\exists j\in\mathbb{Z}_n$: $\mathit{Receive}_{i,j}$(**io** act, **io** chan, **io** cnt, **io** mark)
$\vee$ $\mathit{Inactivate}_i$(**io** act)
$\vee$ Start_i(**io** run, **io** sig, **io** mark, **out** tval, **out** tmark)
$\vee$ $\mathit{Forward}_i$(**in** $\mathtt{act}_i$, **in** $\mathtt{cnt}_i$, **io** sig, **io** mark, **io** tval, **io** tmark, **io** term):
var
(internal variables of the algorithm)
run$\in\mathbb{B}$, cnt$\in\mathbb{Z}_n\to\mathbb{Z}$, sig$\in\mathbb{Z}_n\to\mathbb{B}$, mark$\in\mathbb{Z}_n\to\mathbb{B}$,
tval$\in\mathbb{Z}$, tmark$\in\mathbb{B}$:
$\wedge$ Init(**in** act, **in** chan, **in** term, **in** run, **in** cnt, **in** sig, **in** mark)
$\wedge$ $\Box[\exists i\in\mathbb{Z}_n$: $\mathit{Action}_{n,i}$(**io** act, **io** chan, **io** term, **io** run, **io** cnt, **io** sig, **io** mark, **io** tval, **io** tmark)$]$

a sequence of inductive properties such that in the proof of inductiveness of a property we can use the previously proven properties as additional "axioms". In this way, we can "modularize" the proof by decomposing it into a number of layers that are thin enough to be managed reasonably well.

Core Properties First, we have to make sure that the termination detection algorithm does not unappropriately interfere with the working of the original system. Every behavior of the extended system must be also a possible behavior of the original system; vice versa, every behavior of the original system should be also allowed by the extended system. This idea is expressed by

Proposition 1 (Equivalence) *SystemT* implements *System* and vice versa, i.e., for all n, act, and chan:

$$(\textbf{var}\ \texttt{term}{\in}\mathbb{B}\text{: } \mathit{SystemT}_n(\texttt{act}, \texttt{chan}, \texttt{term})) \Leftrightarrow \mathit{System}_n(\texttt{act}, \texttt{chan}).$$

Proof: *SystemT* is constructed from *System*

1. by adding actions that leave the variables of *System* unchanged and that can be thefore viewed as stuttering steps of *System*, and
2. by conjoining conditions to *Init*, *Send*, and *Receive*.

Any behavior of *SystemT* is therefore a behavior of *System*. In the conditions conjoined to the *System* actions, *SystemT* does not constrain the *System* variables, thus also the converse holds. □

In general, implementation proofs are more complicated because one has to show that every step of the implementation simulates some step of the specification. However, the construction of *SystemT* from *System* makes equivalence such obvious that we content ourselves with above sketch. Because of equivalence, we refer in the context of this section to *SystemT* simply as "the system". The system property that the algorithm is expected to detect is formalized by

Definition 1 (Termination) The system is *terminated* if all processes are inactive and all channels are empty:

$$\begin{aligned}
&\mathit{terminated}_n(\textbf{in}\ \texttt{act}, \textbf{in}\ \texttt{chan}) :\Leftrightarrow \\
&\quad \wedge\ \forall i{\in}\mathbb{Z}_n\text{: } \neg\texttt{act}_i \\
&\quad \wedge\ \forall i{\in}\mathbb{Z}_n, j{\in}\mathbb{Z}_n\text{: } \texttt{chan}_{i,j} = \langle\rangle.
\end{aligned}$$

An important property of termination is established by

Proposition 2 (Stability of Termination) Once the system is terminated, it stays so, i.e. for all n, act, chan:

$$\Box(\mathit{terminated}_n(\texttt{act}, \texttt{chan}) \Rightarrow \Box\mathit{terminated}_n(\texttt{act}, \texttt{chan})).$$

Proof: It suffices to prove

$$\Box(\mathit{terminated}_n(\texttt{act}, \texttt{chan}) \Rightarrow \mathit{terminated}_n(\texttt{act}', \texttt{chan}')).$$

Init implies $\neg terminated_n(\mathtt{act}, \mathtt{chan})$. Assume $terminated_n(\mathtt{act}, \mathtt{chan})$. We know $\neg\mathtt{act}_k$ for every $k{\in}\mathbb{Z}_n$, thus $Send_{i,j}(\mathtt{act}_i, \mathtt{chan})$ is not enabled. From $\mathtt{chan}_{i,j}{=}\langle\rangle$ we know that $Receive_{i,j}(\mathtt{act}, \mathtt{chan}, \mathtt{cnt}, \mathtt{mark})$ is not enabled. By $\neg\mathtt{act}_i$, $Inactivate_i(\mathtt{act})$ does not have any effect. Neither $Start_i(\mathtt{run}, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark})$ nor $Forward_i(\mathtt{act}_i, \mathtt{cnt}_i, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}, \mathtt{term})$ modify $\mathtt{act}$ or $\mathtt{chan}$. $\Box$

The fundamental property of the algorithm is then expressed by

Proposition 3 (Correctness) If the system detects its termination, it has indeed terminated, i.e., for all n, $\mathtt{act}$, $\mathtt{chan}$, $\mathtt{term}$:

$$\begin{array}{l} SystemT_n(\mathtt{act}, \mathtt{chan}, \mathtt{term}) \Rightarrow \\ \quad \Box(\mathtt{term} \Rightarrow terminated_n(\mathtt{act}, \mathtt{chan})). \end{array}$$

The rest of this section is devoted to setting up the stage for finally proving this property.

General Assumption We will establish a number of system invariants that will be used in the proof the invariant derived in the previous section, which in turn will be used to prove the corrrectness property. In order to simplify the formulation of these properties, we will from now on consider arbitrary but fixed n, $\mathtt{act}$, $\mathtt{chan}$, $\mathtt{term}$ (the specification parameters) and take $\mathtt{cnt}$, $\mathtt{sig}$, $\mathtt{mark}$, $\mathtt{tval}$, $\mathtt{tmark}$ (the local variables of the algorithm) such that

$$\begin{array}{l} \wedge\ Init(\mathtt{act}, \mathtt{chan}, \mathtt{term}, \mathtt{run}, \mathtt{cnt}, \mathtt{sig}, \mathtt{mark}) \\ \wedge\ \Box[\exists i{\in}\mathbb{Z}_n\colon Action_{n,i}(\mathtt{act}, \mathtt{chan}, \mathtt{term}, \mathtt{run}, \mathtt{cnt}, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \\ \quad \mathtt{tmark})] \end{array}$$

holds. The following propositions are expressed in the context of this assumption, i.e., they are true only for behaviors of the system.

System Invariants We start the construction of the proof by establishing a simple invariant that assures that ensures that the algorithm is "well-typed".

Proposition 4 (Type Correctness) The system preserves the asserted variable types, i.e.,

$$\begin{array}{l} \Box(\mathtt{act}{\in}\mathbb{Z}_n{\rightarrow}\mathbb{B} \wedge \mathtt{chan}{\in}\mathbb{Z}_n{\times}\mathbb{Z}_n{\rightarrow}\mathbf{Seq}(MSG) \wedge \mathtt{term}{\in}\mathbb{B} \wedge \mathtt{run}{\in}\mathbb{B} \wedge \\ \mathtt{cnt}{\in}\mathbb{Z}_n{\rightarrow}\mathbb{N} \wedge \mathtt{sig}{\in}\mathbb{Z}_n{\rightarrow}\mathbb{B} \wedge \mathtt{mark}{\in}\mathbb{Z}_n{\rightarrow}\mathbb{B} \wedge \mathtt{tval}{\in}\mathbb{N} \wedge \mathtt{tmark}{\in}\mathbb{B}). \end{array}$$

Proof: We show $\Box\mathtt{act}{\in}\mathbb{Z}_n{\rightarrow}\mathbb{B}$. *Init* implies $\mathtt{act}_i{\in}\mathbb{B}$ for every $i{\in}\mathbb{Z}_n$. The only actions that modify $\mathtt{act}$ are *Receive* and *Inactivate* both of which construct $\mathtt{act}'$ from $\mathtt{act}$ by replacing the mapping of some $i{\in}\mathbb{Z}_n$ by an element of $\mathbb{B}$. Type correctness for the other variables is shown analogously. $\Box$

We will not refer to above property explicitly any more but use it implicitly in many of our reasoning steps.

The following proposition guarantees that the algorithm does not make any mistakes when counting messages.

Proposition 5 (Message Count) The system keeps in $\mathtt{cnt}$ track of the number of messages pending in communication channels, i.e.,

$$\Box(msgcnt_n(\textbf{in}\ \texttt{chan},\ \textbf{in}\ \texttt{cnt}))$$

where

$$msgcnt_n(\textbf{in}\ \texttt{chan},\ \textbf{in}\ \texttt{cnt}) :\Leftrightarrow count_n(\texttt{chan}) = \textstyle\sum_{i\in\mathbb{Z}_n}\texttt{cnt}_i,$$
$$count_n(\texttt{chan}) := \textstyle\sum_{i\in\mathbb{Z}_n}\sum_{j\in\mathbb{Z}_n} len(\texttt{chan}_{i,j}),$$
$$len(\texttt{ch}) := \textbf{such}\ l\in\mathbb{N}\colon \exists m_0, \ldots, m_{l-1}\colon \texttt{ch} = \langle m_0, \ldots, m_{l-1}\rangle.$$

Proof: *Init* implies $\texttt{cnt}_i = 0$ and $\texttt{chan}_{i,j} = \langle\rangle$ for every $i\in\mathbb{Z}_n$ and $j\in\mathbb{Z}_n$. Assume $msgcnt_n(\texttt{count}, \texttt{chan})$. We show $msgcnt_n(\texttt{chan}', \texttt{cnt}')$. The only actions that modify chan or cnt are *Send* and *Receive*:

Case $\boldsymbol{Send}_{i,j}$(act$_i$, chan, cnt) We know $count_n(\texttt{chan}') = count_n(\texttt{chan}[(i, j) \mapsto \texttt{chan}_{i,j}\circ\langle m\rangle]) = 1{+}count_n(\texttt{chan}) = 1{+}\sum_{k\in\mathbb{Z}_n}\texttt{cnt}_k = \sum_{k\in\mathbb{Z}_n}\texttt{cnt}[i \mapsto \texttt{cnt}_i{+}1]_k = \texttt{cnt}'$.

Case $\boldsymbol{Receive}_{i,j}$(act, chan, cnt, mark) Analogous to the previous case. □

The following property ensures that Process 0 indeed knows whether there is a token in the system or not.

Proposition 6 (Detection Run) The system records in run whether there is an ongoing termination detection run, i.e.,

$$\Box(\texttt{run} \Leftrightarrow \exists k\in\mathbb{Z}_n\colon \texttt{sig}_k).$$

It turns out that above invariant is itself not inductive; we therefore combine its proof with the proof of

Proposition 7 (Single Token) There is at most one token in the system, i.e.,

$$\Box(\forall k\in\mathbb{Z}_n\colon \texttt{sig}_k \Rightarrow \forall j\in\mathbb{Z}_n\colon (\texttt{sig}_j \Rightarrow j = k)).$$

Proof of Property 6 and of Property 7:

Init implies $\neg\texttt{run}$ and $\neg\texttt{sig}_k$ for every $k\in\mathbb{Z}_n$. Assume $\texttt{run} \Leftrightarrow \exists k\in\mathbb{Z}_n\colon \texttt{sig}_k$ and $\forall k\in\mathbb{Z}_n\colon \texttt{sig}_k \Rightarrow \forall j\in\mathbb{Z}_n\colon (\texttt{sig}_j \Rightarrow j = k)$.

1. We show $\texttt{run}' \Leftrightarrow \exists k\in\mathbb{Z}_n\colon \texttt{sig}'_k$.

   The only actions that modify run or sig are *Start* and *Forward*.

   Case $\boldsymbol{Start}_\mathrm{i}$(run, sig, mark, tval, tmark) By $\texttt{run}'$ and $\texttt{sig}'[i\ominus_n 1]$.

   Case $\boldsymbol{Forward}_i$(act$_i$, cnt$_i$, sig, mark, tval, tmark, term) The action gives $\texttt{sig}_i$ and thus run. Assume $i \neq 0$. Then $\texttt{run}' = \texttt{run}$ and $\texttt{sig}' = \texttt{sig}[i \mapsto \mathsf{false},\ i\ominus_n 1 \mapsto \mathsf{true}]$. Thus we have $\texttt{run}'$ and $\texttt{sig}'[i\ominus_n 1]$. Now assume $i = 0$. Then we have $\neg\texttt{run}'$ and $\texttt{sig}' = \texttt{sig}[i \mapsto \mathsf{false}]$. By $\texttt{sig}_i$ and the induction hypothesis, we thus have $\neg\texttt{sig}'_k$ for every $k\in\mathbb{Z}_n$.

   (Here we used in the proof of Proposition 6 the induction hypothesis of Proposition 7).

2. We show $\forall k \in \mathbb{Z}_n$: $\mathtt{sig'}_k \Rightarrow \forall j \in \mathbb{Z}_n$: $(\mathtt{sig'}_j \Rightarrow j = k)$. The only actions that modify $\mathtt{sig}$ are *Start* and *Forward*.

   Case $\boldsymbol{Start}_i$**(run, sig, mark, tval, tmark)** We know $\neg\mathtt{run}$ and therefore $\neg\mathtt{sig}_j$ for every $j \in \mathbb{Z}_n$. From $\mathtt{sig'} = \mathtt{sig}[i \ominus_n 1 \mapsto \mathsf{true}]$ we then know $\mathtt{sig'}_j \Rightarrow j = i \ominus_n 1$ for every $j \in \mathbb{Z}_n$.

   (Here we used in the proof of Proposition 7 the induction hypothesis of Proposition 6).

   Case $\boldsymbol{Forward}_i$**(act$_i$, cnt$_i$, sig, mark, tval, tmark, term)** The action gives $\mathtt{sig}_i$ and thus $\mathtt{sig}_j \Rightarrow j = i$ for every $j \in \mathbb{Z}_n$. Assume $i \neq \mathsf{0}$. Then $\mathtt{sig'} = \mathtt{sig}[i \mapsto \mathsf{false}, i \ominus_n 1 \mapsto \mathsf{true}]$ and therefore $\mathtt{sig'}_j \Rightarrow j = i \ominus_n 1$ for every $j \in \mathbb{Z}_n$. Now assume $i = \mathsf{0}$. Then $\mathtt{sig'} = \mathtt{sig}[i \mapsto \mathsf{false}]$ and thus $\neg\mathtt{sig}_j$ for every $j \in \mathbb{Z}_n$. $\square$

Now we have set up enough machinery to proof the crucial fact of the termination detection algorithm:

Proposition 8 (Invariant) The system maintains the following invariant:

$$\Box Inv_n(\textsf{in } \mathtt{act}, \textsf{in } \mathtt{cnt}, \textsf{in } \mathtt{sig}, \textsf{in } \mathtt{mark}, \textsf{in } \mathtt{tval}, \textsf{in } \mathtt{tmark})$$

where

$$\begin{array}{l}
Inv_n(\textsf{in } \mathtt{act}, \textsf{in } \mathtt{cnt}, \textsf{in } \mathtt{sig}, \textsf{in } \mathtt{mark}, \textsf{in } \mathtt{tval}, \textsf{in } \mathtt{tmark}) :\Leftrightarrow \\
\quad \forall k \in \mathbb{Z}_n\colon \mathtt{sig}_k \Rightarrow \\
\qquad \vee\ InvC_{n,k}(\textsf{in } \mathtt{act}, \textsf{in } \mathtt{cnt}, \textsf{in } \mathtt{tval}) \\
\qquad \vee\ \mathsf{0} < \mathtt{tval} + \sum_{j \in \mathbb{Z}_n, j \leq k} \mathtt{cnt}_j \\
\qquad \vee\ \exists j \in \mathbb{Z}_n\colon j \leq k \wedge \mathtt{mark}_j \\
\qquad \vee\ \mathtt{tmark}. \\
InvC_{n,k}(\textsf{in } \mathtt{act}, \textsf{in } \mathtt{cnt}, \textsf{in } \mathtt{tval}) :\Leftrightarrow \\
\quad \wedge\ \forall j \in \mathbb{Z}_n\colon j > k \Rightarrow \neg\mathtt{act}_j \\
\quad \wedge\ \mathtt{tval} = \sum_{j \in \mathbb{Z}_n, j > k} \mathtt{cnt}_j
\end{array}$$

Proof: *Init* implies $\neg\mathtt{sig}_k$ for every $k \in \mathbb{Z}_n$. Assume $Inv_n(\mathtt{act}, \mathtt{cnt}, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark})$. We show $Inv_n(\mathtt{act'}, \mathtt{cnt'}, \mathtt{sig'}, \mathtt{mark'}, \mathtt{tval'}, \mathtt{tmark'})$.

Case $\boldsymbol{Send}_{i,j}$**(act$_i$, chan, cnt)** Take $k \in \mathbb{Z}_n$ with sig_k. The only variable in the invariant changed by the action is $\mathtt{cnt}$, thus only the first or the second disjunct of the invariant can be invalidated. However, the action implies $\mathtt{act}_i$ and thus by the invariant also $i < k$; it can therefore not affect the first disjunct. Since $\mathtt{cnt'}$ differs from $\mathtt{cnt}$ only in $\mathtt{cnt'}_i = \mathtt{cnt}_i + 1$, the second disjunct cannot be invalidated either.

(The last remark explains why the second disjunct of the invariant has to establish that the denoted sum remains positive, not just non-zero.)

Case $\boldsymbol{Receive}_{i,j}$**(act, chan, cnt, mark)** Take $k \in \mathbb{Z}_n$ with $\mathtt{sig'}_k$. We have to show

$$\begin{array}{l}
\vee\ \ldots \\
\vee\ \mathsf{0} < \mathtt{tval} + \sum_{l \in \mathbb{Z}_n, l \leq k} \mathtt{cnt}[j \mapsto \mathtt{cnt}_j - 1]_l
\end{array}$$

$\vee\ \exists l{\in}\mathbb{Z}_n\colon\ l \leq k \wedge \mathtt{mark}[j \mapsto \mathsf{true}]_l$
$\vee\ \mathtt{tmark}$

If $j \leq k$, we have $\mathtt{mark}[j \mapsto \mathsf{true}]_j$ and thus the third disjunct.

(This case is the cause of the third disjunct of the invariant: a process which currently holds or did not yet get the token receives a message and becomes marked.)

Assume $j > k$. We know $\mathtt{sig}_k$ from $\mathtt{sig}'_k$ and the action. We proceed by case distinction on the consequence of the induction hypothesis.

Case $(\forall l{\in}\mathbb{Z}_n\colon\ l > k \Rightarrow \neg\mathtt{act}_l) \wedge (\mathtt{tval} = \sum_{l\in\mathbb{Z}_n, l>k}\mathtt{cnt}_l)$ By $j > k$ we have $\sum_{l\in\mathbb{Z}_n, l\leq k}\mathtt{cnt}[j \mapsto \mathtt{cnt}_j{-}1]_l = \sum_{l\in\mathbb{Z}_n, l\leq k}\mathtt{cnt}_l$. The case gives us $\mathtt{tval}{+}\sum_{l\in\mathbb{Z}_n, l\leq k}\mathtt{cnt}_l = \sum_{l\in\mathbb{Z}_n}\mathtt{cnt}_l$. From the action and Proposition 5, we know $\sum_{l\in\mathbb{Z}_n}\mathtt{cnt}_l > 0$ and thus the second disjunct.

(This case is the cause for the second disjunct of the invariant: a message is received by a process that previously held the token, which invalidates the token counter, but keeps the message count positive.)

Case $(0 < \mathtt{tval}{+}\sum_{l\in\mathbb{Z}_n, l\leq k}\mathtt{cnt}_l)$ The second disjunct follows from $j > k$ and thus $\sum_{l\in\mathbb{Z}_n, l\leq k}\mathtt{cnt}_l = \sum_{l\in\mathbb{Z}_n, l\leq k}\mathtt{cnt}[j \mapsto \mathtt{cnt}_j{-}1]_l$.

Case $(l \leq k \wedge \mathtt{mark}_l$**, for some** $l{\in}\mathbb{Z}_n)$ By $j > k$ and $\mathtt{mark}[j \mapsto \mathsf{true}]_l$.

Case $\mathtt{tmark}$ Obvious.

Case $\boldsymbol{Inactivate}_i(\mathtt{act})$ The only effect of this action is to falsify $\mathtt{act}_i$, which cannot invalidate the invariant.

Case $\boldsymbol{Start}_\mathrm{i}(\mathtt{run},\ \mathtt{sig},\ \mathtt{mark},\ \mathtt{tval},\ \mathtt{tmark})$ Take $k{\in}\mathbb{Z}_n$ with $\mathtt{sig}'_k$, therefore we have $\mathtt{sig}[i{\ominus_n}1 \mapsto \mathsf{true}]_k$. By Proposition 7 and $i = 0$, we know $k = i{\ominus_n}1 = n{\ominus_n}1$. We have to show

$\vee\ \wedge\ \forall j{\in}\mathbb{Z}_n\colon\ j > n{\ominus_n}1 \Rightarrow \neg\mathtt{act}_j$
$\quad\ \wedge\ 0 = \sum_{j\in\mathbb{Z}_n, j>n\ominus_n 1}\mathtt{cnt}_j$
$\vee\ \ldots$

which is trivially true.

Case $\boldsymbol{Forward}_i(\mathtt{act}_i,\ \mathtt{cnt}_i,\ \mathtt{sig},\ \mathtt{mark},\ \mathtt{tval},\ \mathtt{tmark},\ \mathtt{term})$ If $i = 0$, we know from $\mathtt{sig}' = \mathtt{sig}[i \mapsto \mathsf{false}]$ that $\neg\mathtt{sig}'_k$ for all $k{\in}\mathbb{Z}_n$. Now assume $i \neq 0$ and take $k{\in}\mathbb{Z}_n$ with $\mathtt{sig}'_k$. Since $\mathtt{sig}' = \mathtt{sig}[i \mapsto \mathsf{false},\ i{\ominus_n}1 \mapsto \mathsf{true}]$, Proposition 7 gives us $k = i{\ominus_n}1$. We have to show

$\vee\ \wedge\ \forall j{\in}\mathbb{Z}_n\colon\ j > i{\ominus_n}1 \Rightarrow \neg\mathtt{act}_j$
$\quad\ \wedge\ \mathtt{tval}{+}\mathtt{cnt}_i = \sum_{j\in\mathbb{Z}_n, j>i\ominus_n 1}\mathtt{cnt}_j$
$\vee\ 0 < \mathtt{tval}{+}\mathtt{cnt}_i{+}\sum_{j\in\mathbb{Z}_n, j\leq i\ominus_n 1}\mathtt{cnt}_j$
$\vee\ \exists j{\in}\mathbb{Z}_n\colon\ j \leq i{\ominus_n}1 \wedge \mathtt{mark}[i \mapsto \mathsf{false}]_j$
$\vee\ \mathtt{tmark} \vee \mathtt{mark}_i.$

The action implies $\mathtt{sig}_i$; we proceed by case distinction on the consequence of the induction hypothesis.

Case $(\forall j{\in}\mathbb{Z}_n\colon\ j > i \Rightarrow \neg\mathtt{act}_j) \wedge (\mathtt{tval} = \sum_{j\in\mathbb{Z}_n, j>i}\mathtt{cnt}_j)$ The action implies $\neg\mathtt{act}_i$ and thus $\forall j{\in}\mathbb{Z}_n\colon\ j > i{\ominus_n}1 \Rightarrow \neg\mathtt{act}_j$. Furthermore we know $\mathtt{tval}{+}\mathtt{cnt}_i = (\sum_{j\in\mathbb{Z}_n, j>i}\ \mathtt{cnt}_j){+}\mathtt{cnt}_i = \sum_{j\in\mathbb{Z}_n, j>i\ominus_n 1}\ \mathtt{cnt}_j$.

Case ($0 < \mathtt{tval}+\sum_{j\in\mathbb{Z}_n, j\le i}\mathtt{cnt}_j$) The second disjunct is a consequence of $\sum_{j\in\mathbb{Z}_n, j\le i}\mathtt{cnt}_j = \mathtt{cnt}_i+\sum_{j\in\mathbb{Z}_n, j\le i\ominus_n 1}\mathtt{cnt}_j$.

Case ($j \le i \wedge \mathtt{mark}_j$, for some $j\in\mathbb{Z}_n$) If $j \le i\ominus_n 1$, we have $\mathtt{mark}[i \mapsto \mathsf{false}]_j$. If $j = i$, we have $\mathtt{mark}_i$.

(The second part of this case is the cause of the fourth disjunct of the invariant. If the token reaches a process that is marked because it has previously received a message, the token gets marked.)

Case $\mathtt{tmark}$ Obvious. □

Conclusion Finally we come to the heart of the matter: showing that our system invariant indeed implies the stated correctness property.

Proof of Proposition 3 (Correctness): *Init* implies $\neg\mathtt{term}$. Assume $\mathtt{term} \Rightarrow \mathit{terminated}_n(\mathtt{act}, \mathtt{chan})$. We show $\mathtt{term}' \Rightarrow \mathit{terminated}_n(\mathtt{act}', \mathtt{chan}')$. If $\mathtt{term}$, then the induction hypothesis and Proposition 2 imply $\mathit{terminated}_n(\mathtt{act}', \mathtt{chan}')$. Now assume $\neg\mathtt{term}$ and $\mathtt{term}'$. We then have $\mathit{Forward}_i(\mathtt{act}_i, \mathtt{cnt}_i, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}, \mathtt{term})$ with $i = 0$. Thus we know:

(1) $\neg\mathtt{act}_0$,
(2) $\mathtt{sig}_0$,
(3) $\mathtt{tval}' = \mathtt{tval}+\mathtt{cnt}_0$,
(4) $\mathtt{tmark}' = \mathtt{tmark} \vee \mathtt{mark}_0$,
(5) $\neg\mathtt{tmark}' \wedge \mathtt{tval}' = 0$.

By (3) and (5), we have

(6) $\mathtt{tval}+\mathtt{cnt}_0 = 0$.

By (4) and (5), we have

(7) $\neg\mathtt{tmark} \wedge \neg\mathtt{mark}_0$.

With (2) and Proposition 8, (6) and (7) give us

(8) $\forall j\in\mathbb{Z}_n\colon j > 0 \Rightarrow \neg\mathtt{act}_j$,
(9) $\mathtt{tval} = \sum_{j\in\mathbb{Z}_n, j>0}\mathtt{cnt}_j$.

(Here we have finally used both parts of the first disjunct of the invariant.)

From (1) and (8), we conclude

(10) $\forall j\in\mathbb{Z}_n\colon \neg\mathtt{act}_j$,

while (6) and (9) give us

(11) $\sum_{j\in\mathbb{Z}_n}\mathtt{cnt}_j = 0$.

Using Proposition 5, $\mathtt{act} = \mathtt{act}'$, and $\mathtt{chan} = \mathtt{chan}'$, (10) and (11) imply $\mathit{terminated}_n(\mathtt{act}', \mathtt{chan}')$. □

Specification 3: Termination Detection with Fairness

$SystemTF_{n\in\mathbb{N}}(\mathtt{act}{\in}\mathbb{Z}_n{\to}\mathbb{B},\ \mathtt{chan}{\in}\mathbb{Z}_n{\times}\mathbb{Z}_n{\to}\mathbf{Seq}(MSG),\ \mathtt{term}{\in}\mathbb{B})\ :\Leftrightarrow$
(A system that eventually detects its termination)
$\ldots$
$\wedge\ \ldots$
$\wedge\ \forall i{\in}\mathbb{Z}_n\text{: } WF(Start_i(\mathtt{run}, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}))$
$\wedge\ \forall i{\in}\mathbb{Z}_n\text{: } WF(Forward_i(\mathtt{act}_i, \mathtt{cnt}_i, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}, \mathtt{term}))$

5 Progress of the Algorithm

Fairness In the previous section, we have shown a so called *safety property* of the termination detection algorithm: if the variable `term` is set, then the system has terminated. We have not yet shown the crucial *progress property* of the algorithm: if the system has terminated, then the algorithm eventually sets `term`. In fact, this property is *not* true for the current specification, because there is a system behavior where the algorithm never sets `term`: the behavior where none of the system variables changes any more (i.e., only stuttering steps are performed after termination).

Of course, we consider this behavior very "unfair" because in a termination state, also either *Start* or *Forward* is continously enabled. We therefore have to augment our algorithm by a condition that prevents such an unfair behavior. We therefore introduce

Definition 2 (Weak Fairness) An action is weakly fair if it is not possible that the action is continuously enabled but never executed:

$$WF(A) :\Leftrightarrow (\Diamond\Box Enabled\ \langle A\rangle) \Rightarrow \Box\Diamond\langle A\rangle.$$

This notion of fairness is called "weak" because it allows an action to be continuedly enabled and disabled without ever being executed. In those cases where the enabling condition of an action continues to be true (as for those of *Signal* or *Forward*), this notion of fairness is however sufficient. The fairness requirements of the termination detection algorithm are then added to our specification as shown in Specification 3.

The new specification is equivalent to the original system:

Proposition 9 (Equivalence) *SystemTF* implements *System* and vice versa, i.e., for all n, `act`, and `chan`:

$$(\mathbf{var}\ \mathtt{term}{\in}\mathbb{B}\text{: } SystemTF_n(\mathtt{act}, \mathtt{chan}, \mathtt{term})) \Leftrightarrow System_n(\mathtt{act},\mathtt{chan}).$$

Proof: Analogous to the Proof of Proposition 1. $\Box$

By construction, for all n, `act`, `chan` and `term`, also $SystemTF_n(\mathtt{act}, \mathtt{chan}, \mathtt{term}) \Rightarrow System_n(\mathtt{act}, \mathtt{chan}\ \mathtt{term})$ holds, but because of the additional fairness constraints the converse is not true.

The crucial progress property of the new specification is then stated by

Proposition 10 (Progress) If the system is terminated, it will eventually detect so, i.e., for all n, `act`, `chan`, `term`:

$$\begin{array}{l} \mathit{SystemTF}_{n\in\mathbb{N}}(\mathtt{act}, \mathtt{chan}, \mathtt{term}) \Rightarrow \\ \quad \Box(\mathit{terminated}_n(\mathtt{act}, \mathtt{chan}) \Rightarrow \Diamond\mathtt{term}). \end{array}$$

Proving Progress In order to show a formula of the form $\Diamond A$, it suffices to show that there exists an "induction term" t depending on the system variables (a *state function*) such that under the assumption $\neg A \wedge \neg A'$

1. t is not increased by any system action,
2. there is some "helpful" system action that decreases t,
3. the helpful action is enabled.

Provided that the denoted ordering does not allow infinitely decreasing sequences of values (i.e., that it is *well-founded*), above conditions ensure that A is eventually executed.

Virtual Variables As already discussed in Section 3, if the system terminates during a termination detection round, it may be necessary to first complete this round and perform two more rounds until termination can be concluded. The induction term therefore depends on the information in which round the termination detection algorithm is. Unfortunately, this information is not yet captured in any of the existing program variables. We therefore introduce a "virtual" variable

$$\mathtt{trun} \in \mathbb{Z}_3$$

which is initialized to 2

$$\begin{array}{l} \mathit{Init}(\ldots) :\Leftrightarrow \\ \quad \wedge \ \ldots \\ \quad \wedge \ \mathtt{trun} = 2, \end{array}$$

and decreased by Process 0 at the end of a termination detection round, if the system has terminated: the variable is set to 1, if some other process in the system is still marked and to 0, otherwise. Thus trun describes an upper bound for the number of detection rounds to be performed until termination can be definitely concluded.

$$\begin{array}{l} \mathit{Forward}_i(\ldots) :\Leftrightarrow \\ \quad \wedge \ \ldots \\ \quad \wedge \ \mathtt{trun}' = \\ \qquad \quad \textbf{if } i \neq 0 \vee \neg \mathit{terminated}_n(\mathtt{act}, \mathtt{chan}) \textbf{ then } \mathit{trun} \\ \qquad \qquad \textbf{else if } \exists j \in \mathbb{Z}_n \colon j > 0 \wedge \mathtt{mark}_j \textbf{ then } 1 \textbf{ else } 0 \end{array}$$

Specification 4: Introduction of A Virtual Variable

$SystemTFV_{n\in\mathbb{N}}(\texttt{act}\in\mathbb{Z}_n\rightarrow\mathbb{B},\ \texttt{chan}\in\mathbb{Z}_n\times\mathbb{Z}_n\rightarrow\textbf{Seq}(MSG),\ \texttt{term}\in\mathbb{B})$:⇔
  (A system that eventually detects its termination)
  let
    $Init$(**in** act, **in** chan, **in** term, **in** run, **in** cnt, **in** sig, **in** mark, **in** trun) :⇔
      ∧ ...
      ∧ trun = 2,
    ...
    $Forward_i$(**in** a, **in** c, **io** sig, **io** mark, **io** tval, **io** tmark, **io** term, **io** trun)
    :⇔
      ∧ ...
      ∧ $\texttt{trun}' = \textbf{if}\ i \neq 0 \lor \neg terminated_n(\texttt{act}, \texttt{chan})\ \textbf{then}\ trun$
          $\textbf{else if}\ \exists j\in\mathbb{Z}_n\colon j > 0 \land \texttt{mark}_j\ \textbf{then}\ 1\ \textbf{else}\ 0$
    $Action_{n,i}$(**io** act, **io** chan, **io** term, **io** run, **io** cnt, **io** sig, **io** mark, **io** tval, **io** tmark, **io** trun) :⇔
      ∨ ...
      ∨ $Forward_i(\texttt{act}_i, \texttt{cnt}_i$, sig, mark, tval, tmark, term, trun):
  var
    ...
    $\texttt{trun}\in\mathbb{Z}_3$:
  ∧ $Init$(act, chan, term, run, cnt, sig, mark, trun)
  ∧ $\Box[\exists i\in\mathbb{Z}_n\colon Action_{n,i}$(act, chan, term, run, cnt, sig, mark, tval, tmark, trun)]
  ∧ ...

Of course, this behavior cannot be implemented (otherwise it would already solve the termination detection problem!); however, the only purpose of this variable is to give us some means for expressing the induction term to be constructed for proving progress of the algorithm. See Specification 4 for a complete description of the extension.

We have to check that the new variable is indeed virtual, i.e., does not at all influence the original program behavior:

Proposition 11 (Equivalence) *SystemTFV* implements *SystemTF* and vice versa, i.e., for all n, act, chan, and term:

$$SystemTFV_n(\texttt{act}, \texttt{chan}, \texttt{term})) \Leftrightarrow SystemTF_n(\texttt{act}, \texttt{chan}, \texttt{term}).$$

Proof: trun is not used except for computing trun′. □

As in the previous section, we assume fixed n, act, chan, term and all internal variables such that all conditions in the specification hold. The following propositions are therefore only true for behaviors of *SystemTFV* ("the system").

As usual, we immediately assert some type condition for the new variable trun.

Proposition 12 (Type Correctness) The system preserves the type of trun.

$$\Box(\texttt{trun}\in\mathbb{Z}_3).$$

Proof: *Init* implies trun = 2. Only *Forward* sets trun by assigining 1 or 0. □

Induction Term Based on $\mathtt{trun}$, we now define the induction term that we expect to decrease with every action of the terminated system.

Definition 3 (Induction Term) We define the induction term

$$num_n(\mathtt{sig}, \mathtt{trun}) := (\mathtt{trun}, tpos_n(\mathtt{sig}))$$

where

$$\begin{array}{l} tpos_n(\mathtt{sig}) := \\ \quad \textsf{such } i{\in}\mathbb{Z}_{n+1}\text{:} \\ \quad\quad \vee\ \forall j{\in}\mathbb{Z}_n\text{: } \neg\mathtt{sig}_j \wedge i = n \\ \quad\quad \vee\ \exists j{\in}\mathbb{Z}_n\text{: } \mathtt{sig}_j \wedge i = j. \end{array}$$

The first component of *num* denotes the number of detection rounds to be performed, the second component the number of processes that the token still has to pass in the current round.

The domain of the induction term is stated by

Proposition 13 (Type Correctness) *num* is well typed:

$$\Box(num_n(\mathtt{sig}, \mathtt{trun}){\in}\mathbb{Z}_3{\times}\mathbb{Z}_{n+1}).$$

Proof: By definition and Proposition 12. □

We then have to assert that *num* is indeed a suitable induction term.

Proposition 14 (Well Foundedness) The domain $(\mathbb{Z}_3{\times}\mathbb{Z}_{n+1}, \prec)$ with

$$\begin{array}{l} (t_0, p_0) \prec (t_1, p_1) :\Leftrightarrow \\ \quad \vee\ t_0 < t_1 \\ \quad \vee \wedge\ t_0 = t_1 \\ \quad\quad \wedge\ p_0 < p_1 \end{array}$$

has not infinitely decreasing sequences of elements.

Proof: Obvious. □

System Invariants The following invariants characterize $\mathtt{trun}$.

Proposition 15 (Invariant) If $\mathtt{trun}$ has been decreased, the system has terminated:

$$\Box(\mathtt{trun} < 2 \Rightarrow terminated_n(\mathtt{act}, \mathtt{chan})).$$

Proof: *Init* implies $\mathtt{trun} = 2$. Assume $\mathtt{trun} < 2 \Rightarrow terminated_n(\mathtt{act}, \mathtt{chan})$ and $\mathtt{trun}' < 2$. If $\mathtt{trun} < 2$, we have $terminated_n(\mathtt{act}, \mathtt{chan})$ and Proposition 2 implies $terminated_n(\mathtt{act}', \mathtt{chan}')$. If $\mathtt{trun} = 2$, we have $Forward_0(\mathtt{act}_i, \mathtt{cnt}_i, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}, \mathtt{term}, \mathtt{trun})$ and, by the action, $terminated_n(\mathtt{act}, \mathtt{chan})$. Proposition 2 then implies $terminated_n(\mathtt{act}', \mathtt{chan}')$. □

Proposition 16 (Invariant) If $\mathtt{trun}$ has been decreased, the token value reflects the actual message count:

$$\Box(\texttt{trun} < 2 \Rightarrow (\forall k \in \mathbb{Z}_n\colon \texttt{sig}_k \Rightarrow \texttt{tval} = \textstyle\sum_{j\in\mathbb{Z}_n, j>k} \texttt{cnt}_j)).$$

Proof: *Init* implies $\texttt{trun} = 2$. Assume $\texttt{trun} < 2 \Rightarrow \ldots$ and $\texttt{trun}' < 2$. If $\texttt{trun} = 2$, we have $\mathit{Forward}_0(\texttt{act}_0, \texttt{cnt}_i, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark}, \texttt{term}, \texttt{trun})$ and thus $\neg\texttt{sig}'_k$ for all $k\in\mathbb{Z}_n$. Assume $\texttt{trun} < 2$ and take $k\in\mathbb{Z}_n$with $\texttt{sig}'_k$. By Proposition 15, we have $\mathit{terminated}_n(\texttt{act}, \texttt{chan})$, thus *Send* and *Receive* are not enabled and *Inactivate* has no effect. $\mathit{Start}_i(\texttt{run}, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark})$ implies $\texttt{tval}' = 0$; Proposition 7 gives $k = n-1$. $\mathit{Forward}_i(\texttt{act}_i, \texttt{cnt}_i, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark}, \texttt{term}, \texttt{trun})$ implies $\texttt{tval}' = \texttt{tval}+\texttt{cnt}_i$. If $i > 0$, Proposition 7 yields $k = i-1$; otherwise, we have $\neg\texttt{sig}'_k$ for every $k\in\mathbb{Z}_n$. $\Box$

Proposition 17 (Invariant) If trun has been decreased once, all processes that previously held the token are unmarked:

$$\Box(\texttt{trun} = 1 \Rightarrow (\forall k\in\mathbb{Z}_n\colon \texttt{sig}_k \Rightarrow (\forall j\in\mathbb{Z}_n\colon j > k \Rightarrow \neg\texttt{mark}_j))).$$

Proof: *Init* implies $\texttt{trun} = 2$. Assume $\texttt{trun} = 1 \Rightarrow \ldots$ and $\texttt{trun}' = 1$. If $\texttt{trun} \neq 1$, we have $\mathit{Forward}_0(\texttt{act}_0, \texttt{cnt}_i, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark}, \texttt{term}, \texttt{trun})$ and thus $\neg\texttt{sig}'_k$ for all $k\in\mathbb{Z}_n$. Assume $\texttt{trun} = 1$ and take $k\in\mathbb{Z}_n$with $\texttt{sig}'_k$. By Proposition 15, we have $\mathit{terminated}_n(\texttt{act}, \texttt{chan})$, thus *Send* and *Receive* are not enabled and *Inactivate* has no effect. $\mathit{Start}_i(\texttt{run}, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark})$ and Proposition 7 imply $k = n-1$. $\mathit{Forward}_i(\texttt{act}_i, \texttt{cnt}_i, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark}, \texttt{term}, \texttt{trun})$ implies $\neg\texttt{mark}'_i$. If $i > 0$, Proposition 7 yields $k = i-1$, otherwise we have $\neg\texttt{sig}'_k$ for every $k\in\mathbb{Z}_n$. $\Box$

Proposition 18 (Invariant) If trun has been decreased twice, all processes are unmarked and the token is unmarked in a detection run:

$$\begin{aligned}&\Box(\texttt{trun} = 0 \Rightarrow\\ &\quad\wedge\ \forall k\in\mathbb{Z}_n\colon \neg\texttt{mark}_k\\ &\quad\wedge\ \texttt{run} \Rightarrow \neg\texttt{tmark}).\end{aligned}$$

Proof: *Init* implies $\texttt{trun} = 2$. Assume $\texttt{trun} = 0 \Rightarrow \ldots$ and $\texttt{trun}' = 0$. If $\texttt{trun} \neq 0$, we have $\mathit{Forward}_0(\texttt{act}_0, \texttt{cnt}_i, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark}, \texttt{term}, \texttt{trun})$, which implies $\neg\texttt{run}'$ and (by the definition of $\texttt{trun}'$ and of $\texttt{mark}'$) $\neg\texttt{mark}'_k$ for every $k\in\mathbb{Z}_n$. Assume $\texttt{trun} = 0$ and take $k\in\mathbb{Z}_n$. By Proposition 15, we have $\mathit{terminated}_n(\texttt{act}, \texttt{chan})$, thus *Send* and *Receive* are not enabled and *Inactivate* has no effect. As well $\mathit{Start}_i(\texttt{run}, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark})$ as $\mathit{Forward}_i(\texttt{act}_i, \texttt{cnt}_i, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark}, \texttt{term}, \texttt{trun})$ imply with the hypothesis $\neg\texttt{tmark}'$ and $\neg\texttt{mark}'_k$ for every $k\in\mathbb{Z}_n$. $\Box$

Conclusion Finally we combine all our knowledge to the

Proof of Proposition 10 (Progress): Assume $\mathit{terminated}_n(\texttt{act}, \texttt{chan})$ and $\neg\texttt{term}$ and $\neg\texttt{term}'$. We show that

1. $\mathit{num}_n(\texttt{sig}, \texttt{term})$ is not increased (with respect to $\prec$) by any action,
2. $\mathit{num}_n(\texttt{sig}, \texttt{term})$ is decreased by the helpful action

   $\vee\ \mathit{Start}_i(\texttt{run}, \texttt{sig}, \texttt{mark}, \texttt{tval}, \texttt{tmark})$

$\vee$ $\mathit{Forward}_i(\mathtt{act}_i, \mathtt{cnt}_i, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}, \mathtt{term}, \mathtt{trun})$,

3. this action is enabled.

From the weak fairness condition of the decreasing action and Propositions 13 and 14 we can then conclude $\Diamond\mathtt{term}$.

1. The only actions that affect `sig` and `trun` are *Start* and *Forward*.

2. **Case $\boldsymbol{\mathit{Start}_\mathrm{i}}$(run, sig, mark, tval, tmark)** The action implies $\neg\mathtt{run}$; Proposition 6 gives us $\mathit{num}_n(\mathtt{sig}, \mathtt{trun}) = (\mathtt{trun}, n)$. The action implies $\mathit{num}_n(\mathtt{sig}', \mathtt{trun}') = (\mathtt{trun}, n{-}1)$.

   Case $\boldsymbol{\mathit{Forward}_i}$(act$_i$, cnt$_i$, sig, mark, tval, tmark, term, trun) We have $\mathtt{sig}_i$; Proposition 7 implies $\mathit{num}_n(\mathtt{sig}, \mathtt{trun}) = (\mathtt{trun}, i)$. If $i \neq 0$, we have $\mathtt{trun}' = \mathtt{trun}$ and $\mathtt{sig}'_{i-1}$; Proposition 7 then yields $\mathit{num}_n(\mathtt{sig}', \mathtt{term}') = (\mathtt{trun}, i{-}1)$. If $i = 0$, $\neg\mathtt{term}'$ implies

   $\vee$ $\mathtt{tmark} \vee \mathtt{mark}_0$
   $\vee$ $\mathtt{tval}{+}\mathtt{cnt}_0 \neq 0$

   By $\mathit{terminated}_n(\mathtt{act}, \mathtt{chan})$, Proposition 5, and Proposition 16, we then have $\mathtt{tval}{+}\mathtt{cnt}_0 = 0$. Above disjunction and Proposition 18 thus give $\mathtt{trun} \neq 0$.

   Case $\mathtt{trun} = 1$ By Proposition 17, $\forall j{\in}\mathbb{Z}_n\colon j > 0 \Rightarrow \neg\mathtt{mark}_j$. Therefore we have $\mathtt{trun}' = 0$.

   Case $\mathtt{trun} = 2$ By $\mathit{terminated}_n(\mathtt{act}, \mathtt{chan})$, we have $\mathtt{trun}' < 2$.

3. If `run`, then $\mathit{terminated}_n(\mathtt{act}, \mathtt{chan})$ and Proposition 6 ensure that $\mathit{Forward}_i(\mathtt{act}_i, \mathtt{cnt}_i, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}, \mathtt{term}, \mathtt{trun})$ is enabled for some $i{\in}\mathbb{Z}_n$. If $\neg\mathtt{run}$, then $\mathit{Start}_0(\mathtt{run}, \mathtt{sig}, \mathtt{mark}, \mathtt{tval}, \mathtt{tmark}, \mathtt{t})$ is enabled. $\Box$

References

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning, Greenwich, Conneticut, 1996.

[Dij98] Edsger W. Dijkstra. Shmuel Safra's Version of Termination Detection (Note EWD998). In *Proofs and Calculations*, Working Material for the International Summer School on Calculational System Design, Marktoberdorf, Germany, July 28–August 9, 1998. Advanced Study Institute.

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