ELSEVIER Science of Computer Programming 24 (1995) 105-128

Science of Computer Programming

Verifying programs that use causally-ordered message-passing

Scott D. Stoller *,1, Fred B. Schneider 2

Department of Computer Science, Cornell University, Ithaca, NY 14853, USA

Received May 1994; revised January 1995
Communicated by K.R. Apt

Abstract

We give an operational model of causally-ordered message-passing primitives. Based on this model, we formulate a Hoare-style proof system for causally-ordered delivery. To illustrate the use of this proof system and to demonstrate the feasibility of applying invariant-based verification techniques to algorithms that depend on causally-ordered delivery, we verify an asynchronous variant of the distributed termination detection algorithm of Dijkstra, Feijen, and van Gasteren.

1. Introduction

Causally-ordered delivery can be understood as a generalization of FIFO ordering [21]. In both, a message is delivered only after all messages on which it may depend. With FIFO ordering, this guarantee applies only to messages having the same sender; with causal ordering, this guarantee applies to messages sent by any process. Additional motivation for and examples of the use of causally-ordered delivery can be found in [3].

This paper gives a proof system for causally-ordered delivery. Our proof system is similar in style to the satisfaction-based logics for synchronous message-passing in [15], for ordinary asynchronous message-passing in [19], and for flush channels in [6]. We assume familiarity with the terminology of that literature.

* Corresponding author.
1 Supported by an IBM Graduate Fellowship.
2 Supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant CCR-9003440, DARPA/NSF Grant No. CCR-9014363, NASA/DARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

Reasoning about message-passing primitives for causally-ordered delivery involves a global property, the system-wide causality relation, which defines what messages are deliverable. This distinguishes causally-ordered delivery from the types of message passing for which axiomatic semantics have already been given (e.g., [9,15,19,6]). Our work demonstrates that substantially new methods are not required when message-delivery semantics depends on global information.

A program proof in a satisfaction-based logic involves discharging three obligations:

(1) a proof outline characterizes execution of each process in isolation,

(2) a "satisfaction proof" validates postconditions of receive statements, and

(3) an interference-freedom proof establishes that execution of no process invalidates an assertion in another.

Our proof system for causally-ordered message-passing is similar, except step (2) is merged with step (1). (Such a merging is also possible for other satisfaction-based proof systems that handle asynchronous communication primitives, like the logics of [19] and [6].)

The remainder of the paper is organized as follows. Section 2 defines causally-ordered message-passing. Our proof system is the subject of Section 3. In Section 4, we use the proof system to verify an asynchronous variant of the distributed termination detection algorithm of Dijkstra, Feijen, and van Gasteren [7]. Section 5 contains some conclusions.

2. A model of causally-ordered message-passing

We give an operational semantics for causally-ordered message-passing primitives by translating programs containing these primitives into a generic concurrent programming language that has shared variables. The shared variables represent the state of the network.

Processes communicate by sending and receiving messages. To encode the restrictions implicit in causally-ordered delivery, each message sent is modeled in our translation by a triple $\langle d,i,t\rangle$, where[3]

d is the data being sent by the program,

i is the name of the process[4] that sent the message, and

t is a *timestamp* that contains information used to determine whether the message is ready for delivery.

The following functions are useful in connection with messages represented by triples.

$$data(\langle d,i,t\rangle) \triangleq d$$

$$sender(\langle d,i,t\rangle) \triangleq i$$

[3] An actual implementation of causally-ordered delivery might not require a sender name i or timestamp t. That information is used here to abstract from the details of all real implementations.

[4] Processes are named $0, 1, \ldots, N-1$, and hereafter identifiers i, j, k, and p range over process names.

$$ts(\langle d,i,t\rangle) \overset{\Delta}{=} t$$

Two shared variables σ_i and ρ_i are associated with each process i. Variable σ_i contains the (triples modeling) messages sent to process i; ρ_i contains the (triples modeling) messages process i has received. These variables have two roles: they model the current state of the network and they record the communications history. Thus, each serves both as a history variable and a special program variable — variables are *special* in the sense that they may be read or written only by executing send and receive statements.

There is an obvious and seemingly simpler alternative to using variables σ_i and ρ_i. It is to use a single variable χ_i (say), where the value of χ_i is the set of messages sent to process i but not yet received (i.e., χ_i equals $\sigma_i - \rho_i$). The model we use has two advantages over this one-variable model. First, in our model, proving interference freedom (defined in Section 3) is easier. This is because no process can falsify $m \in \sigma_i$ or $m \in \rho_i$; predicate $m \in \chi_i$ would be invalidated by the receiver. Second, proofs of some programs (such as the example in Section 4) involve reasoning about communications history. That history is available in σ_i and ρ_i but is not available in χ_i. The use of special variables for recording communications history was first proposed for Gypsy [1].

Causally-ordered delivery restricts when a message can be received. This is achieved in our translation by defining a well-founded partial order $\prec$ on timestamps. Our definition of $\prec$ is based on the theory of [13]. A system execution is represented as a tuple of sequences of events; each sequence corresponds to the execution of a single process. An *event* is a *send* event, a *receive* event, or an *internal* (i.e., non-communication) event. The *happens-before* (or "potential causality") relation $\rightarrow$ for a system execution is the smallest transitive binary relation on the events in that execution such that:

- If e and e' are performed by the same process and e occurs before e', then $e \rightarrow e'$.
- If e is the send event for a message m and e' is the receive event for that message, then $e \rightarrow e'$.

Causally-ordered delivery is formalized in terms of $\rightarrow$ as follows [4].[5] Let *send*(m) and *receive*(m) respectively denote the send event and receive event for a message m.

> **Causally-ordered Delivery**: If m and m' are sent to the same process and *send*$(m) \rightarrow$ *send*(m'), then *receive*$(m) \rightarrow$ *receive*(m').[6]

To illustrate this definition, consider the system of three processes illustrated in Fig. 1. Process 1 sends a message $m_{1,3}$ (say, an update to a database record) to process 3 and then sends a message $m_{1,2}$ to process 2 (informing it of the update). After receiving $m_{1,2}$, process 2 sends a message $m_{2,3}$ (containing an update that depends on the update in $m_{1,2}$) to process 3. Since *send*$(m_{1,3}) \rightarrow$ *send*$(m_{2,3})$, causally-ordered delivery ensures

[5] To be consistent with the definition in [13], clause (2) of the definition of $\rightarrow$ on page 278 of [4] should be $\forall m$: *send*$(m) \rightarrow$ *deliver*(m). Here, *deliver* denotes the event referred to in our paper as *receive*.

[6] FIFO delivery can also be formalized in terms of $\rightarrow$. FIFO delivery ensures that if m and m' are sent by the same process, to the same process, and *send*$(m) \rightarrow$ *send*(m'), then *receive*$(m) \rightarrow$ *receive*(m'). The close analogy between FIFO delivery and causally-ordered delivery should now be evident.

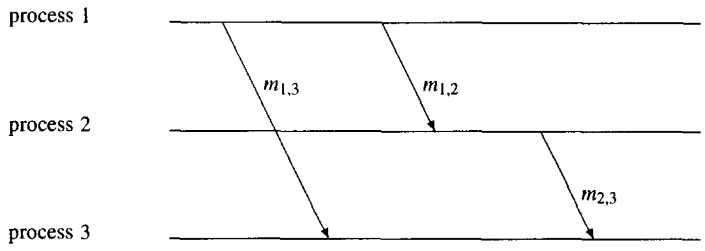

Fig. 1. Example of causally-ordered delivery.

that process 3 receives $m_{1,3}$ before $m_{2,3}$ (as desired). FIFO delivery would not ensure this.

To implement Causally-ordered Delivery using timestamped messages, the timestamps and $\prec$ are chosen to satisfy

$$ts(m) \prec ts(m') \text{ iff } send(m) \rightarrow send(m'). \tag{1}$$

Causally-ordered Delivery is then equivalent to requiring that a message m' is received by a process p only after p has received all messages m sent to p for which $ts(m) \prec ts(m')$ holds.

One way to achieve (1) is to use vector clocks [8,16]. Here, a vector vt_i of type array$[0..N-1]$ of Nat is associated with process i, where vt_i satisfies:

Vector Clock Property: $vt_i[j]$ is the number of send events that are performed by process j and causally precede the next event to be performed by process i.

Partial order $\prec$ is defined in terms of vector clocks, as follows.

$$vt_1 \neq vt_2 \triangleq (\exists i\colon\ vt_1[i] \neq vt_2[i])$$
$$vt_1 \prec vt_2 \triangleq (\forall i\colon\ vt_1[i] \leqslant vt_2[i]) \wedge vt_1 \neq vt_2$$

Three rules define how the vt_i are updated in order to maintain the Vector Clock Property. Since only send events and receive events are of interest, vector clocks are updated only when send and receive statements are executed. Let $inc(vt, i)$ denote a vector vt with the ith component incremented by one. The rules are:

Initialization Rule: Initially, $vt_i[j] = 0$ for all i and j.

Send Update Rule: When process i sends a message m, it updates vt_i by executing

$$vt_i := inc(vt_i, i)$$

and includes updated vector vt_i as the timestamp attached to m.

Receive Update Rule: When a process i receives a message m, it updates vt_i by executing

$$vt_i := \max(vt_i, ts(m)),$$

where $\max(vt, vt')$ is the component-wise maximum of the vectors vt and vt'.

See [16] for an explanation of why these rules ensure that (1) holds.

We now give our translation of send and receive statements into statements that read and write shared variables σ_i and ρ_i. The following notation is used to describe the multiple-assignment [10] of e_1 to x_1, e_2 to x_2, ..., and e_n to x_n:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} := \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix}$$

A send statement **send** e **to** i in process j is translated into:

$$\begin{pmatrix} vt_j \\ \sigma_i \end{pmatrix} := \begin{pmatrix} inc(vt_j, j) \\ \sigma_i \oplus \langle e, j, inc(vt_j, j) \rangle \end{pmatrix} \tag{2}$$

where $s \oplus x \triangleq s \cup \{x\}$.

The translation of a receive statement requires a conditional delay. Statement **await** B **then** S delays until B holds and then executes S as a single indivisible operation starting from a state that satisfies B. A receive statement **receive** x in process i delays until a message is available for receipt and then updates x, ρ_i, and vt_i. In particular, to ensure causally-ordered delivery, **receive** x delays until there exists some message m that has been sent to i, m has not been received, and all messages m' that have been or will be sent to i for which $ts(m') \prec ts(m)$ have been received.

For a set A of triples modeling messages, $choose(A)$ and $minset(A)$ are assumed to satisfy:

$$choose(A) \in A \quad \text{provided } A \neq \emptyset \tag{3}$$

$$minset(A) \triangleq \{m \in A \mid (\forall m' \in A\colon \neg(ts(m') \prec ts(m)))\} \tag{4}$$

A receive statement **receive** x in process i is translated as follows, where m_i is a fresh variable.

$$\begin{aligned} \textbf{await } \sigma_i - \rho_i \neq \emptyset \textbf{ then } & m_i := choose(minset(\sigma_i - \rho_i)) \\ & x := data(m_i) \\ & vt_i := \max(vt_i, ts(m_i)) \\ & \rho_i := \rho_i \oplus m_i \end{aligned} \tag{5}$$

First, note that code fragments (2) and (5) correctly encode the Send Update Rule and Receive Update Rule, so (1) holds. To show that code fragments (2) and (5) correctly implement Causally-ordered Delivery, consider some message m that is received by a process i. We must show that no message m' subsequently received by process i satisfies $send(m') \longrightarrow send(m)$. Suppose such a message m' exists. By (1), $ts(m') \prec ts(m)$. Message m' could not be in σ_i when m is received, since m is selected from among the elements of σ_i with minimal timestamps. Thus, m' must be added to σ_i after m is received. We show that this is impossible by proving: For all messages m and m', if m' is added to σ_i after m has been added, then $\neg(ts(m') \prec ts(m))$.

First, observe that the following holds throughout execution of a program.

$$(\forall j,k\colon\ vt_j[k] \leqslant vt_k[k] \wedge (\forall m \in \sigma_j\colon\ ts(m)[k] \leqslant vt_k[k])) \tag{6}$$

Initially, this holds because for all j and k, $vt_j[k] = 0$ and $\sigma_j = \emptyset$. Only send and receive statements change the values of these variables, so it suffices to show that our translations of these statements preserve (6), which is easily done.

Finally, we show that $\neg(ts(m') \prec ts(m))$. This is implied by $(\exists k\colon\ ts(m')[k] > ts(m)[k])$, which, in turn, follows from $ts(m')[j] > ts(m)[j]$ where j is the sender of m'. The latter holds because $ts(m')[j] = vt_j[j] + 1 > vt_j[j] \geqslant ts(m)[j]$, where the equality follows from the translation of send statements, the strict inequality follows from standard arithmetic, and the nonstrict inequality follows from (6).

3. Axioms for send and receive

We now present Hoare-style axioms [12] for the send and receive statements described above.

Given the above translation of **send** e **to** i into a multiple-assignment statement, we use the multiple-assignment axiom [10] to obtain an axiom for the send statement. The notation $e[x_1 := e_1, \ldots, x_n := e_n]$ denotes the simultaneous substitution of each term e_i for the corresponding variable x_i in a term e. Validity of the following triple follows immediately from the multiple-assignment axiom:

$$\{P[vt_j := inc(vt_j, j),\ \sigma_i := \sigma_i \oplus \langle e, j, inc(vt_j, j)\rangle]\}$$
$$\begin{pmatrix} vt_j \\ \sigma_i \end{pmatrix} := \begin{pmatrix} inc(vt_j, j) \\ \sigma_i \oplus \langle e, j, inc(vt_j, j)\rangle \end{pmatrix}$$
$$\{P\}$$

Thus, we have

Send Axiom: For a send statement in process j:

$$\{P[vt_j := inc(vt_j, j),\ \sigma_i := \sigma_i \oplus \langle e, j, inc(vt_j, j)\rangle]\}\ \textbf{send}\ e\ \textbf{to}\ i\ \{P\} \tag{7}$$

An inference rule for receive statements is obtained using translation (5) of **receive** x. Using axiom (3) for *choose*, the usual rules for assignment and sequential composition, and the inference rule for **await** statements [17]

Await Rule:

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}} \tag{8}$$

we can show that $\{P\}$ **receive** x $\{Q\}$ is valid iff the following Predicate Logic formula is valid:

$$P \wedge m_i \in minset(\sigma_i - \rho_i)$$
$$\Rightarrow Q[x := data(m_i),\ vt_i := \max(vt_i, ts(m_i)),\ \rho_i := \rho_i \oplus m_i].$$

Thus, the inference rule for receive statements is

Receive Rule: For a receive statement in process i:

$$\frac{\begin{array}{l} P \wedge m_i \in minset(\sigma_i - \rho_i) \\ \quad \Rightarrow Q[x := data(m_i),\ vt_i := \max(vt_i, ts(m_i)),\ \rho_i := \rho_i \oplus m_i] \end{array}}{\{P\}\ \textbf{receive}\ x\ \{Q\}} \tag{9}$$

Causally-ordered delivery is encoded in the hypothesis of this rule. Specifically, the assumption that m_i is causally minimal among unreceived messages sent to the receiving process embodies the assumption of causally-ordered delivery.

Interference freedom

The preceding rules for send and receive, together with rules for other statements and the usual miscellaneous rules of Hoare logics (e.g., the Rule of Consequence), can be used to construct a proof outline for each process in isolation. A *proof outline* is a program annotated with an assertion before and after every statement. A proof outline characterizes the behavior of a process assuming that no other process invalidates assertions in that proof outline. The proof outlines for processes that execute concurrently are combined to obtain a proof outline for the entire system by showing *interference freedom* [17] — that no process invalidates assertions in the proof outline of another process.

In a proof outline *PO*, the assertion that precedes a statement S is called the *precondition* of S and is denoted $pre(S)$, the assertion that follows a statement S is called the *postcondition* of S and is denoted $post(S)$, and we write $pre(PO)$ and $post(PO)$ to denote the first and last assertions, respectively, in *PO*. We write $\{P\}\ PO\ \{Q\}$ to denote the triple obtained by changing $pre(PO)$ to P and $post(PO)$ to Q.

An assertion P appearing in a proof outline PO_i is *interference free* with respect to proof outlines $PO_1, \ldots, PO_N$ if for all assignments, sends, and receives S in proof outlines other than PO_i,

$$\{P \wedge pre(S)\}\ S\ \{P\} \tag{10}$$

is valid. This is because (10) asserts that execution of S does not invalidate P. Assignment to variables is the only way to invalidate an assertion.[7] Since our translations for send and receive contain assignments, the interference freedom obligations require checking (10) for each send and receive statement, as well as for each assignment to an ordinary program variable.

[7] This is actually an assumption about the assertion language. For example, it rules out allowing control predicates in assertions. This restriction is not essential. It can be removed by generalizing the definition of interference freedom. We have opted for the simpler theory here, because control predicates are orthogonal to the subject of this paper, namely, modeling causally-ordered delivery.

Proof outlines $PO_1, \dots, PO_N$ are *interference free* if all assertions P in the proof outlines are interference free in $PO_1, \dots, PO_N$. This leads to the following inference rule.

Parallel Composition Rule:

$$\frac{PO_1, \dots, PO_N \qquad PO_1, \dots, PO_N \text{ are interference free}}{\{\bigwedge_i pre(PO_i)\}\ [PO_1 \parallel \dots \parallel PO_N]\ \{\bigwedge_i post(PO_i)\}} \tag{11}$$

Note that, in contrast to the logics for asynchronous communication in [19] and [6], our parallel composition rule does not have a "satisfaction" obligation. This is not an artifact of causally-ordered message-passing; the logics of [19] and [6] could be similarly formulated.

4. Example: distributed termination detection

To illustrate our proof rules, we give a proof outline for the termination detection algorithm of [7]. Validity of this proof outline shows that the algorithm correctly detects quiescence in systems of processes that communicate using causally-ordered message-passing. Our proof outline is based on the correctness argument given in [7], modified for causally-ordered delivery instead of the synchronous communication assumed there.

The algorithm is intended for use in systems where processes behave as follows. At each instant, a process is either *active* or *quiescent*, where the only action possible by a quiescent process is receipt of a message. A quiescent process may become active upon receipt of a message; an active process becomes quiescent spontaneously. Each process i has the form

```
$Init_i$
do
  $\mathop{[]}\limits_{j \neq i}$   $g_{ij}$          $\longrightarrow$ send $e_{ij}$ to $j$
                         $S_{ij}$
  $[]$   receive $x_i$  $\longrightarrow$ $R_i$
od                                                    (12)
```

where the g_{ij} are boolean expressions, and $Init_i$, S_{ij}, and R_i are statements that do not contain communication statements. A process i is quiescent iff each guard g_{ij} is *false*. This is formalized by:

$$q_i \triangleq \neg\left(\bigvee_j g_{ij}\right)$$

In the algorithm of [7], a token circulates among the processes. This introduces a new kind of message, which we call a *token message*. To distinguish it from the messages in the original computation, hereafter called *basic messages*, we use a predicate *istok*(*data*(m)) that holds exactly when m is a token message. Note that a process of the form (12) cannot send basic messages to itself.[8] Define:

$$\sigma_i^{tok} \triangleq \{m \in \sigma_i \mid istok(data(m))\}$$

$$\rho_i^{tok} \triangleq \{m \in \rho_i \mid istok(data(m))\}$$

$$\chi_{i,j} \triangleq \{m \in \sigma_j - \rho_j \mid \neg istok(data(m)) \wedge sender(m) = i\}$$

The system is quiescent if every process is quiescent and no messages are in transit. Thus, the system is quiescent iff the following predicate Q holds.

$$Q \triangleq (\forall i\colon\ q_i \wedge (\forall j\colon\ \chi_{i,j} = \emptyset))$$

Code for the detection algorithm appears in Fig. 2. Angle brackets indicate that the enclosed statement is executed atomically [14].[9] Our goal is to prove that Q holds when process 0 reaches the statement in $RELAY_0$ preceded by the comment "quiescent". Thus, we must construct a proof outline in which the precondition of that statement implies Q. The program in Fig. 2 does not take any special action when quiescence is detected — process 0 simply executes **skip**. A round of communication could easily be added to notify each process that quiescence has been detected.

We now describe in detail how the algorithm works, presenting side-by-side informal explanations and corresponding formal assertions.

A color, either black or white, is associated with each process. For each process i, we introduce a boolean variable b_i such that b_i is *true* iff process i is black. The detection algorithm sets b_i to *true* when process i sends a basic message; its sets b_i to *false* when i sends a token message. Therefore, we can assert that b_i holds if process i has a sent a basic message since it last sent a token message. This is formalized as an assertion in terms of the following state function:[10]

lx_i: The largest timestamp in $\{m \in \bigcup_j \sigma_j^{tok} \mid sender(m) = i\}$, if such a timestamp exists; otherwise $\vec{0}$.

The assertion about b_i is now formalized as:

$$J_1 \triangleq (\forall i\colon\ (\exists j\colon\ (\exists m \in \chi_{i,j}\colon\ lx_i \prec ts(m))) \Rightarrow b_i)$$

The algorithm proceeds as a sequence of rounds. One process serves as the initiator for all rounds; it starts each round by sending a token message. Without loss of generality, assume process 0 is the initiator. In each round, the token is received by every process exactly once, ending with the initiator. We define the token to be *at position i* if it has

[8] This restriction is not needed for correctness of the algorithm; we adopt it here because it simplifies the correctness proof slightly.

[9] Angle brackets are not necessary for correctness. We use them because they eliminate the need for control predicates (or auxiliary variables) in the proof. Introducing control predicates requires additional proof rules and a more general definition of interference freedom. This would lengthen the correctness proof, so for simplicity of exposition, we use angle brackets.

[10] The name lx_i is a mnemonic for "last transmission" of the token by process i.

```
Code for Process $i$

    $INIT_i$
    do
    $\underset{j \neq i}{[]}\ g_{ij}$      $\longrightarrow$ $b_i := true$
                        send $e_{ij}$ to $j$
                        $S_{ij}$
    $[]$ receive $y_i$ $\longrightarrow$ if $istok(y_i)$   $\longrightarrow$ $\langle h_i := true$
                                             $t_i := tokval(y_i)\rangle$
                        $[]$ $\neg istok(y_i)$ $\longrightarrow$ $x_i := y_i$
                                             $R_i$
                        fi
    $[]$ $q_i \wedge h_i$     $\longrightarrow$ $RELAY_i$
    od

$INIT_0$   $\triangleq$ send $mktok(false)$ to $N-1$
           $Init_0$

$RELAY_0$ $\triangleq$ if $(t_0 \vee b_0)$   $\longrightarrow$ $\langle$send $mktok(false)$ to $N-1$
                                   $h_0 := false$
                                   $b_0 := false\rangle$
           $[]$ $\neg(t_0 \vee b_0)$ $\longrightarrow$ (* quiescent *)
                                   skip
           fi

For $0 < j < N$:

    $INIT_j$   $\triangleq$ $Init_j$

    $RELAY_j$ $\triangleq$ $\langle$send $mktok(t_j \vee b_j)$ to $j-1$
               $h_j := false$
               $b_j := false\rangle$
```

Fig. 2. Termination Detection Algorithm.

been sent to process i and not subsequently sent by process i; we say that the token *visits* a process when the token has been received by but not sent from that process. For each process i, we introduce a new variable h_i that is *true* iff the token is visiting process i.

In each round, the token visits the processes in descending order by process name. Thus, the token visits process $N-1$, $N-2$, ..., 0, and the current *t*oken *p*osition is given by the state function:

$$tp \triangleq \begin{cases} i-1 & \text{if } (\forall j \neq i\colon\ lx_j \prec lx_i) \\ N-1 & \text{otherwise} \end{cases}$$

Note that all arithmetic on process names is modulo N.

An assertion J_{tok} says that the token circulates among the processes in a certain fixed order. J_{tok} states that the N most recent sends of token messages are totally ordered by causality. This is equivalent to stipulating that timestamps on these token messages form an ascending sequence. For example, if $tp \neq N-1$, then $lx_{tp} \preceq lx_{tp-1} \preceq \cdots \preceq lx_0 \preceq lx_{N-1} \preceq lx_{N-2} \preceq \cdots \preceq lx_{tp+1}$, where $vt_1 \preceq vt_2 \triangleq vt_1 \prec vt_2 \vee vt_1 = vt_2$. Formally,

$$J_{tok} \triangleq (\forall i \neq tp\colon\ lx_{i+1} \preceq lx_i)$$

Note that the truth of J_{tok} does not depend on the use of causally-ordered message-passing.

An assertion relating the timestamps of token messages to the timestamps of basic messages is also needed. For this, we use an assertion J_{bas}, whose informal interpretation is as follows.

Let m be a basic message sent from i to k that was sent before the αth transmission of the token by i. If m was sent in the same direction that the token travels (i.e., if $k < i$), then m must be delivered before the αth transmission of the token by receiver k. If m was sent in the other direction (i.e., if $i \leqslant k$), then m must be delivered before the $(\alpha+1)$st transmission of the token by receiver k.

The use of causally-ordered message-passing is essential for ensuring that J_{bas} holds throughout execution of the algorithm. The validity of J_{bas} also depends on other properties of the algorithm. For example, it depends on J_{tok}, as one can see from the proof of T9 in the Appendix. Informally, to conclude that the system is quiescent, we must show (among other things) that all basic messages have been delivered. J_{bas} enables us to do this by providing an upper bound on when basic messages must be delivered. The bound is expressed in terms of the timestamps of token messages.

We formalize the assertion using an additional state function.

nlx_i: The second largest timestamp in $\{m \in \bigcup_j \sigma_j^{tok} \mid sender(m) = i\}$, if such a timestamp exists; otherwise $\vec{0}$.

$$\begin{aligned} J_{bas} \triangleq (\forall i,k\colon\ \forall m \in \chi_{i,k}\colon\ & \\ & (k \leqslant tp < i && \Rightarrow nlx_i \prec ts(m)) \\ \wedge\ & (k < i \wedge \neg(k \leqslant tp < i) && \Rightarrow lx_i \prec ts(m)) \\ \wedge\ & (i \leqslant tp < k && \Rightarrow lx_i \prec ts(m)) \\ \wedge\ & (i \leqslant k \wedge \neg(i \leqslant tp < k) && \Rightarrow nlx_i \prec ts(m))) \end{aligned} \tag{13}$$

Assertions J_1, J_{bas}, and J_{tok} contain all of the information about message-delivery order needed for correct operation of the algorithm. We encapsulate this information as a single assertion J:

$$J \triangleq J_1 \wedge J_{bas} \wedge J_{tok}$$

As with processes, a color, either black or white, is associated with the token. The color of the token is represented as before — black is encoded as *true*, and white is encoded as *false*. While in transit, this boolean value is included in each token message; while the token is visiting a process i, a new variable t_i is used to store the color of the last token message received by process i.

Given a boolean value c, $mktok(c)$ denotes a token value whose color is c. The color of the token is extracted using a selector *tokval*. Thus, $istok(mktok(c)) = true$ and $tokval(mktok(c)) = c$ hold. In each round, the token is initially white. It becomes black (if it is not already) when it visits a process i (i.e. h_i equals *true*) that is black (i.e. b_i equals *true*). Thus, the token becomes black when it visits a process that has sent a basic message since last sending a token message, and the current *t*oken *c*olor is given by:

$$tc \triangleq \begin{cases} t_{tp} \vee b_{tp} & \text{if } h_{tp} \\ false & \text{if } \neg h_{tp} \wedge (\exists m \in \sigma_{tp}^{tok}\colon\ ts(m) = lx_{tp+1} \\ & \qquad\qquad\qquad \wedge\ tokval(data(m)) = false) \\ true & \text{otherwise} \end{cases}$$

We also associate with each process i a variable y_i, which is used for temporary storage of received values.

When the token returns to the initiator, if either the initiator or the token is black, then the initiator starts another round. If both are white, then the system is quiescent (i.e., Q holds). Informally, when a white token returns to the initiator, we conclude that at the time of the token's last visit to each process, that process had not sent a basic message since the token's previous visit to that process. From this and J_{bas}, we know that all basic messages sent to each process were delivered before the token's final visit to that process, hence no process was re-activated after the token's final visit. Thus, all processes are quiescent and all channels are empty — the system must be quiescent. This is reflected in the proof outlines of Fig. 3 by the Q in the precondition for the second branch of the alternation statement $RELAY_0$.

The operation of the algorithm is succinctly characterized by K, where $K \triangleq K_1 \vee K_2 \vee K_3$ and

$$K_1 \triangleq (\forall i > tp\colon\ q_i \wedge (\forall k\colon\ \chi_{i,k} = \emptyset)) \wedge (h_{tp} \Rightarrow (\forall k \geqslant tp\colon\ \chi_{tp,k} = \emptyset))$$

$$K_2 \triangleq (\exists i \leqslant tp\colon\ b_i)$$

$$K_3 \triangleq tc$$

K_1 says that every process visited by the token in the current round is quiescent and no basic message sent by one of these processes is in transit. Moreover, if the token is visiting process tp, then no basic messages sent by process tp are in transit to processes the token has visited in this round. Informally, K_1 implies that every process visited by the token in the current round is quiescent and will remain quiescent unless one of them receives a message from a process not yet visited by the token in the current round. K_2 says that some process not already visited by the token during the current round is black. Informally, if some such process is black, then it might have sent a message that will re-activate a process already visited by the token during the current round. Finally, K_3 says that the token is black. Informally, the token is black if it visited a process that recently sent messages; these messages could re-activate other processes, so the system might not be quiescent.

Assertions J and K are not quite strong enough to prove correctness of the algorithm. An assertion I that expresses several simple properties of the algorithm (e.g., that there is always at most one token message in the system) is also needed. Thus, we define $\mathcal{I} \triangleq I \wedge J \wedge K$, where

$$
\begin{aligned}
I \triangleq (\forall i:\ & (|\{i \mid \sigma_i^{tok} \neq \rho_i^{tok}\}| \leqslant 1) && (14)\\
& \wedge (|\sigma_i^{tok} - \rho_i^{tok}| \leqslant 1) && (15)\\
& \wedge (\forall m \in \sigma_i\colon\ ts(m) \preceq vt_{sender(m)}) && (16)\\
& \wedge (\forall m \in \sigma_i\colon\ ts(m) \preceq vt_i \Rightarrow m \in \rho_i) && (17)\\
& \wedge (\forall m \in \rho_i\colon\ ts(m) \preceq vt_i) && (18)\\
& \wedge ((h_i \vee \sigma_i^{tok} \neq \rho_i^{tok}) \Rightarrow tp = i) && (19)\\
& \wedge (h_i \Rightarrow (\sigma_i^{tok} \neq \emptyset \wedge (\forall j\colon\ \sigma_j^{tok} = \rho_j^{tok}))) && (20)\\
& \wedge (\sigma_i^{tok} = \{m \in \cup_j \sigma_j^{tok} \mid sender(m) = i+1\}) && (21)\\
& \wedge (total(\{m \in \cup_j \sigma_j \mid sender(m) = i\})) && (22)\\
& \wedge (total(\cup_j \sigma_j^{tok})) && (23)\\
& \wedge (lx_i \preceq vt_i) && (24)\\
& \wedge (\chi_{i,i} = \emptyset)) && (25)
\end{aligned}
$$

and $total(S) \triangleq (\forall m, m' \in S\colon\ m \neq m' \Rightarrow ts(m) \prec ts(m') \vee ts(m') \prec ts(m))$.

Conjuncts (14) and (15) together say that at most one token message is in transit. Conjunct (16) says that the vector time of a process is greater than or equal to the timestamp of every message previously sent by that process. Conjunct (17) says that messages whose timestamps precede or equal the vector time of their destination have been received. Conjunct (18) says that the vector time of a process is greater than or equal to the timestamps of all messages it has received. Conjunct (19) says that $tp = i$ if the token is visiting process i (i.e., h_i holds) or if the token has been sent to but not yet received by process i. Conjunct (20) says that when h_i holds, the token was sent to process i and no token messages are in transit. Conjunct (21) says that only process $i+1$ sends token messages to process i. Conjunct (22) says that the timestamps of the messages sent by process i are totally ordered. Conjunct (23) says that the timestamps of all token messages ever sent are totally ordered. Conjunct (24) says that the timestamp of the last token message sent by process i is less than or equal to the vector time of process i. Conjunct (25) says that no basic messages are in transit from process i to itself.

Proof outlines for processes augmented to detect termination appear in Fig. 3. The Appendix contains a detailed justification of the proof outlines.

Communication statements may appear in guards, so we use the following proof rule for iteration statements:

Iteration Rule:

$$\frac{\text{For } i \in [1..N],\quad \{I \wedge g_i\}\ C_i\ \{P_i\}\ PO_i\ \{I\}}{\begin{array}{l} \{I\} \\ \textbf{do} \\ \quad \underset{i\in[1..N]}{[\!]}\ g_i; C_i \longrightarrow \{P_i\} \\ \qquad\qquad\qquad\quad PO_i \\ \qquad\qquad\qquad\quad \{I\} \\ \textbf{od} \\ \left\{ I \wedge \neg\left(\bigvee_{i\in[1..N]} g_i\right)\right\} \end{array}} \tag{26}$$

Here, g_i is a boolean expression and C_i is a receive or skip statement.[11] One might expect there to be an assertion between g_i and C_i in the rule's conclusion. Expression g_i contains program variables of only process i, so g_i cannot be invalidated by execution of another process. In particular, interference cannot occur even if evaluation of g_i and execution of C_i are not performed as a single indivisible action. Thus, there is no need to make the assertion explicit.

To illustrate reasoning about receive statements, we give a detailed proof for the triple

$$\begin{array}{l} \{\mathcal{I}\} \\ \textbf{receive } y_i \\ \{\mathcal{I} \wedge (\neg istok(y_i) \Rightarrow K[q_i := false]) \\ \quad \wedge\ (istok(y_i) \Rightarrow tp = i \wedge \neg h_i \wedge tc = tokval(y_i))\} \end{array} \tag{27}$$

This triple arises as a hypothesis in the application of the Iteration Rule to the main loop of each process. The triple expresses a crucial fact about the algorithm — that activation of a process (i.e., the changing of q_i to false) by reception of a basic message does not falsify K. By Receive Rule (9), we can deduce (27) from

$$\begin{array}{l} \mathcal{I} \wedge m_i \in \sigma_i - \rho_i \\ \quad \Rightarrow (\mathcal{I} \wedge (\neg istok(y_i) \Rightarrow K[q_i := false]) \\ \qquad\quad \wedge\ (istok(y_i) \Rightarrow tp = i \wedge \neg h_i \wedge tc = tokval(y_i)))[\theta] \end{array} \tag{28}$$

where the substitution θ is:

$$y_i := data(m_i),\ vt_i := \max(vt_i, ts(m_i)),\ \rho_i := \rho_i \oplus m_i$$

We show in the Appendix that $\mathcal{I} \Rightarrow \mathcal{I}[\theta]$ is valid. Here, we first show that

$$\mathcal{I} \wedge m_i \in \sigma_i - \rho_i \wedge \neg istok(y_i[\theta]) \Rightarrow K[q_i := false][\theta] \tag{29}$$

[11] The guard "g; **skip**" is abbreviated "g"; the guard "*true*; **receive** x" is abbreviated "**receive** x".

Proof Outline for Process *i*

```
$\{\mathcal{I} \wedge \neg h_i \wedge tp \geqslant i \wedge (i = 0 \Rightarrow (\forall j\colon\ \sigma_j^{tok} = \emptyset))\}$
$INIT_i$   $\{\mathcal{I}\}$
do
$\mathop{[\,]}\limits_{j \neq i}$ $g_{ij}$          $\longrightarrow$ $\{\mathcal{I} \wedge g_{ij}\}$
                           $b_i := true$   $\{\mathcal{I} \wedge g_{ij} \wedge b_i\}$
                           send $e_{ij}$ to $j$   $\{\mathcal{I} \wedge g_{ij}\}$
                           $S_{ij}$   $\{\mathcal{I}\}$
[] receive $y_i$ $\longrightarrow$ $\{\mathcal{I} \wedge (\neg istok(y_i) \Rightarrow K[q_i := false])$
                              $\wedge (istok(y_i) \Rightarrow tp = i \wedge \neg h_i \wedge tc = tokval(y_i))\}$
                           if $istok(y_i)$    $\longrightarrow$ $\{\mathcal{I} \wedge tp = i \wedge \neg h_i \wedge tc = tokval(y_i)\}$
                                                 $\langle h_i := true$
                                                  $t_i := tokval(y_i)\rangle$   $\{\mathcal{I}\}$
                           [] $\neg istok(y_i)$ $\longrightarrow$ $\{\mathcal{I} \wedge K[q_i := false]\}$
                                                 $x_i := y_i$   $\{\mathcal{I} \wedge K[q_i := false]\}$
                                                 $R_i$   $\{\mathcal{I}\}$
                           fi   $\{\mathcal{I}\}$
[] $q_i \wedge h_i$      $\longrightarrow$ $\{\mathcal{I} \wedge q_i \wedge h_i\}$
                           $RELAY_i$   $\{\mathcal{I}\}$
od
$\{\mathcal{I}\}$
```

```
$INIT_0$   $\triangleq$ send $mktok(false)$ to $N-1$   $\{\mathcal{I} \wedge \neg h_0 \wedge tp \geqslant 0\}$
           $Init_0$
```

```
$RELAY_0$ $\triangleq$ if  $(t_0 \vee b_0)$    $\longrightarrow$ $\{\mathcal{I} \wedge h_0\}$
                                    $\langle$send $mktok(false)$ to $N-1$
                                     $h_0 := false$
                                     $b_0 := false\rangle$   $\{\mathcal{I}\}$
             [] $\neg(t_0 \vee b_0)$ $\longrightarrow$ $\{\mathcal{I} \wedge Q\}$
                                    (* quiescent *)
                                    skip   $\{\mathcal{I}\}$
           fi
```

For $0 < j < N$:

```
$INIT_j$    $\triangleq$ $Init_j$

$RELAY_j$ $\triangleq$ $\langle$send $mktok(t_j \vee b_j)$ to $j-1$
              $h_j := false$
              $b_j := false\rangle$
```

Fig. 3. Proof outlines.

is valid. We assume the antecedent and prove the consequent. Note that

$$K[q_i := \mathit{false}][\theta] = (K_1[\theta][q_i := \mathit{false}] \vee K_2 \vee K_3)$$

Thus, if K_2 or K_3 holds, then so does (29). Suppose neither K_2 nor K_3 holds. Since $\mathcal{I}$ holds by assumption, K must also hold, so K_1 must hold as well. We now show that in this case, $K_1[\theta][q_i := \mathit{false}]$ holds. First, note that $K_1[\theta]$ holds; this follows easily from the fact that K_1 holds. The proof proceeds by case analysis on the relative values of i and tp.

case $i \leqslant tp$**.** $K_1[\theta]$ does not depend on the q_j's for $j \leqslant tp$. Therefore, since $K_1[\theta]$ holds, so does $K_1[\theta][q_i := \mathit{false}]$.

case $i > tp$**.** We show that this case is impossible. Let $k \triangleq \mathit{sender}(m_i)$. From the antecedent of (29) and the definition of $\chi_{k,i}$, we conclude $m_i \in \chi_{k,i}$.

case $k \leqslant tp$**.** Instantiating the universally quantified variables i and k in J_{bas} with k and i, respectively, we conclude (using the third conjunct of J_{bas}) that $lx_k \prec ts(m_i)$. Using J_1, this implies that b_k holds, which implies that K_2 holds. This contradicts the assumption above that neither K_2 nor K_3 hold.

case $k > tp$**.** By assumption, K_1 holds, so $(\forall j\colon\ \chi_{k,j} = \emptyset)$, so $\chi_{k,i} = \emptyset$. From the antecedent of (29), we have $\neg \mathit{istok}(y_i[\theta])$ (i.e., $\neg \mathit{istok}(\mathit{data}(m_i))$) and $m_i \in \sigma_i - \rho_i$, so by definition of $\chi_{k,i}$, we have $m_i \in \chi_{k,i}$, a contradiction.

Finally, consider showing that $(\mathit{istok}(y_i) \Rightarrow tp = i \wedge \neg h_i \wedge tc = \mathit{tokval}(y_i))[\theta]$ holds whenever the antecedent of (28) holds. This is equivalent to showing

$$\mathcal{I} \wedge m_i \in \sigma_i - \rho_i \wedge \mathit{istok}(\mathit{data}(m_i)) \Rightarrow tp = i \wedge \neg h_i \wedge tc = \mathit{tokval}(\mathit{data}(m_i)) \quad (30)$$

We assume the antecedent and prove the consequent. From the antecedent, we conclude $m_i \in \sigma_i^{tok} - \rho_i^{tok}$. Thus, $\sigma_i^{tok} \neq \rho_i^{tok}$, so by conjunct $(\forall i\colon\ (h_i \vee \sigma_i^{tok} \neq \rho_i^{tok}) \Rightarrow tp = i))$ in I, $tp = i$ holds. We next show, by contradiction, that $\neg h_i$ holds. Suppose not; then h_i holds, so (using I), $\sigma_i^{tok} = \rho_i^{tok}$, which contradicts $m_i \in \sigma_i^{tok} - \rho_i^{tok}$. Finally, we show that $tc = \mathit{tokval}(\mathit{data}(m_i))$. From I, $|\sigma_i^{tok} - \rho_i^{tok}| \leqslant 1$; thus, m_i is the only unreceived message in σ_i^{tok}, so m_i must have the largest timestamp in σ_i^{tok}, and $ts(m_i) = lx_{i+1}$. If $\mathit{tokval}(\mathit{data}(m_i)) = \mathit{false}$, then the second clause in the definition of tc applies, so $tc = \mathit{false}$; otherwise, $\mathit{tokval}(\mathit{data}(m_i)) = \mathit{true}$, and by conjunct $\mathit{total}(\cup_j \sigma_j^{tok})$ in I, no other message in σ_i^{tok} has timestamp lx_{i+1}, so the second clause in the definition of tc does not apply, so by the third clause in that definition, $tc = \mathit{true}$. Thus, in either case, $tc = \mathit{tokval}(\mathit{data}(m_i))$.

Comparison to related work

This distributed termination detection algorithm was first presented in [7] for systems that use synchronous communication. Apt [2] formalized the partial-correctness argument of [7] and proved some additional properties of the algorithm. Verjus then attempted to give another proof of partial-correctness [22], but his argument was flawed

[20]. This suggests that showing correctness of this algorithm is non-trivial, even if synchronous communication is being assumed.

The first correctness argument applicable to this algorithm in an asynchronous setting is (to the best of our knowledge) an operational argument due to Raynal and Helary [18]. Proposition 3.8.1 in [18] establishes partial correctness assuming that the message-delivery order satisfies a property P. Our proof assumes causally-ordered delivery, which implies our predicate J_{bas}; J_{bas} is similar to but slightly stronger than property P of [18].

Another operational (albeit more formal) proof, by Charron-Bost et al., appears in [5]. It shows correctness of this termination detection algorithm for systems that communicate using causally-ordered message-passing. The proofs there differ considerably from the invariant-based analysis of the synchronous case in [7]. In fact, Charron-Bost et al. claim that correctness proofs for all algorithms that use causally-ordered delivery "must consider the execution as a whole, rather than concentrate on assertions that remain invariant in each global state" ([5, p. 34]). The existence of our proof, which is an invariant-based analysis, refutes this claim.

5. Conclusions

We have presented a Hoare-style proof system for causally-ordered delivery. Through an example, we have demonstrated the feasibility of our approach to reasoning about causally-ordered delivery. The example, a distributed termination detection algorithm, has been treated using other approaches, so there is now an opportunity to compare those approaches with the one in this paper.

The fact that a correctness proof for causally-ordered delivery can be based closely on the analysis of a synchronous version is a significant benefit of the approach discussed in this paper. We support a two-step approach to verifying algorithms that use asynchronous message-passing [11]:

1. Verify a synchronous version of the algorithm (presumably a simpler task).
2. Modify the algorithm and the proof to obtain a correctness proof for the asynchronous version of the algorithm.

One benefit of this two-step approach is that it leads naturally to a focus on and accurate determination of the ordering requirements needed by the algorithm. An interesting question is the extent to which this approach can be made formal and systematic.

Appendix A. Proof of correctness

We show that the proof outlines in Fig. 3 are valid. We discuss only the triples for non-composite statements. It is easy to prove validity of the proof outlines in Fig. 3 using these results and the inference rules for sequential composition, iteration, and alternation. The triples for non-composite statements that arise in the proofs for each

For $0 < i < N$:

T1 : $\{\mathcal{I} \wedge \neg h_i \wedge tp \geqslant i\}$ $Init_i$ $\{\mathcal{I}\}$
T2 : $\{\mathcal{I} \wedge g_{ij}\}$ $b_i := true$ $\{\mathcal{I} \wedge g_{ij} \wedge b_i\}$
T3 : $\{\mathcal{I} \wedge g_{ij} \wedge b_i\}$ **send** e_{ij} **to** j $\{\mathcal{I} \wedge g_{ij}\}$
T4 : $\{\mathcal{I} \wedge g_{ij}\}$ S_{ij} $\{\mathcal{I}\}$
T5 : $\{\mathcal{I}\}$ **receive** y_i $\{\mathcal{I} \wedge (\neg istok(y_i) \Rightarrow K[q_i := false])$
$\wedge (istok(y_i) \Rightarrow tp = i \wedge \neg h_i \wedge tc = tokval(y_i))\}$
T6 : $\{\mathcal{I} \wedge tp = i \wedge \neg h_i \wedge tc = tokval(y_i)\}$ $\langle h_i := true \quad t_i := tokval(y_i)\rangle$ $\{\mathcal{I}\}$
T7 : $\{\mathcal{I} \wedge K[q_i := false]\}$ $x_i := y_i$ $\{\mathcal{I} \wedge K[q_i := false]\}$
T8 : $\{\mathcal{I} \wedge K[q_i := false]\}$ R_i $\{\mathcal{I}\}$
T9 : $\{\mathcal{I} \wedge q_i \wedge h_i\}$ $\langle$**send** $mktok(t_i \vee b_i)$ **to** $i-1 \quad h_i := false \quad b_i := false\rangle$ $\{\mathcal{I}\}$
T10 : $\{\mathcal{I} \wedge \neg h_0 \wedge tp \geqslant 0 \wedge (\forall j\colon \sigma_j^{tok} = \emptyset)\}$
send $mktok(false)$ **to** $N-1$ $\{\mathcal{I} \wedge \neg h_0 \wedge tp \geqslant 0\}$
T11 : $\{\mathcal{I} \wedge \neg h_0 \wedge tp \geqslant 0\}$ $Init_0$ $\{\mathcal{I}\}$
T12 : $\{\mathcal{I} \wedge h_0\}$ $\langle$**send** $mktok(false)$ **to** $N-1 \quad h_0 := false \quad b_0 := false\rangle$ $\{\mathcal{I}\}$

Fig. 4. Triples for non-composite statements.

process in isolation are listed in Fig. 4. Proving invariance of I is straightforward, so we omit those details. For brevity, we sometimes are content with an informal explanation for why a triple is valid; based on this, the reader should have little difficulty constructing a formal proof.

A.1. Proof for process $i > 0$ in isolation

T1 : $\{\mathcal{I} \wedge \neg h_i \wedge tp \geqslant i\}$ $INIT_i$ $\{\mathcal{I}\}$

Since $i > 0$, $INIT_i$ is $Init_i$. J is unaffected by execution of $Init_i$ because $Init_i$ neither sends nor receives messages. To see that K is also unaffected, note that the only variables that appear in K and can be assigned by $Init_i$ are those appearing in q_i, and that K is independent of q_i for $i \leqslant tp$. The precondition of T1 implies $i \leqslant tp$, so K is not invalidated by $INIT_i$.

T2 : $\{\mathcal{I} \wedge g_{ij}\}$ $b_i := true$ $\{\mathcal{I} \wedge g_{ij} \wedge b_i\}$

J is unaffected by execution of this statement. Variable b_i occurs only positively in K, so setting b_i to true never falsifies K. Finally, b_i does not appear in g_{ij}, so the assignment to b_i does not falsify g_{ij}.

T3 : $\{\mathcal{I} \wedge g_{ij} \wedge b_i\}$ **send** e_{ij} **to** j $\{\mathcal{I} \wedge g_{ij}\}$

We prove invariance of J as follows. J_1 is preserved because b_i holds. J_{tok} is unaffected because the message being sent is not a token message. Let m denote the element added to σ_j by executing this statement. To show that J_{bas} is preserved, it suffices to show that $lx_i \prec ts(m)$ and $nlx_i \prec ts(m)$ hold, since J_{bas} is then satisfied regardless of which conjunct applies to this message. By definition of the send statement, $ts(m) = inc(vt_i, i)$, so (by definition of $\prec$) $vt_i \prec ts(m)$. From I, we have $lx_i \preceq vt_i$, so by transitivity of $\prec$, $lx_i \prec ts(m)$. It follows from the definitions of lx_i and nlx_i that $nlx_i \preceq lx_i$, so by transitivity of $\prec$, $nlx_i \prec ts(m)$. Thus, J_{bas} is preserved.

The proof that K is preserved is by case analysis on the disjunct of K that holds initially.

case K_1. In this case, $tp \geqslant i$ must also hold, since $i > tp$ and K_1 imply q_i, contradicting g_{ij} in the precondition of T3. Since $i \leqslant tp$ and b_i hold, K_2 also holds, so see that case.

case K_2. K_2 is unaffected by execution of this statement, so K_2 still holds after execution of this statement.

case K_3. K_3 is unaffected by execution of this statement, so K_3 still holds after execution of this statement.

T4 : $\{\mathcal{I} \wedge g_{ij}\}\ S_{ij}\ \{\mathcal{I}\}$

J is unaffected by execution of S_{ij} because S_{ij} neither sends nor receives messages. The only variables that appear in K and can be assigned by S_{ij} are those appearing in q_i. Since g_{ij} holds, q_i is false, so execution of S_{ij} either truthifies q_i or leaves it unchanged. Variable q_i occurs only positively in K, so truthifying q_i never falsifies K.

T5 : $\{\mathcal{I}\}$ **receive** $y_i\ \{\mathcal{I} \wedge (\neg istok(y_i) \Rightarrow K[q_i := false])$
$\wedge (istok(y_i) \Rightarrow tp = i \wedge \neg h_i \wedge tc = tokval(y_i))\}$

Adding elements to ρ_i never falsifies J or K, and J and K do not depend on y_i or vt_i, so J and K are preserved by execution of this statement. We argued in Section 4 that the other conjuncts in the postcondition hold after execution of this statement.

T6 : $\{\mathcal{I} \wedge tp = i \wedge \neg h_i \wedge tc = tokval(y_i)\}\ \langle h_i := true \quad t_i := tokval(y_i)\rangle\ \{\mathcal{I}\}$

J is unaffected by execution of this statement because messages are neither sent nor received. The proof that K is preserved is by case analysis on the disjunct of K that holds initially. Note that the only variables or state functions appearing in K that are affected by execution of this statement are tc and h_{tp}.

case K_1. The first conjunct of K_1 is unaffected by execution of this statement. We now consider the second conjunct. If $(\forall k \geqslant i\colon\ \chi_{i,k} = \emptyset)$, then, since $tp = i$ appears in the precondition, we can conclude that K_1 holds after h_i is set to *true* by this statement. If $(\forall k \geqslant i\colon\ \chi_{i,k} = \emptyset)$ does not hold, then there exist k and m such that $k \geqslant i$ and $m \in \chi_{i,k}$. I implies $\chi_{i,i} = \emptyset$, so it must be that $k > i$ and $m \in \chi_{i,k}$. From the precondition of this triple, $i = tp$, so $i \leqslant tp < k$. Thus, by the third conjunct of J_{bas}, $lx_i \prec ts(m)$, so by J_1, b_i holds. Since $tp = i$ and b_i hold, K_2 must hold, so see that case.

case K_2. K_2 is unaffected by execution of this statement, so K_2 still holds after execution of this statement.

case K_3. In this case, tc holds. Execution of this statement changes tc from $tokval(y_i)$ to $tokval(y_i) \vee b_i$, so K_3 is not falsified.

T7 : $\{\mathcal{I} \wedge K[q_i := false]\}\ x_i := y_i\ \{\mathcal{I} \wedge K[q_i := false]\}$

J is unaffected by execution of this statement because messages are neither sent nor received. Note that x_i can appear in K only in q_i. Since $K[q_i := false]$ holds before execution, and since q_i occurs only positively in K, changing q_i can't falsify K. Finally, $K[q_i := false]$ is unaffected by execution of this statement.

T8 : $\{\mathcal{I} \wedge K[q_i := false]\}\ R_i\ \{\mathcal{I}\}$

J is unaffected by execution of this statement because messages are neither sent nor received. The only variables that appear in K and can be assigned by R_i are those appearing in q_i. Since q_i occurs only positively in K, and since K holds even if q_i doesn't (because $K[q_i := false]$ appears in the precondition), execution of this statement cannot falsify K.

T9 : $\{\mathcal{I} \wedge q_i \wedge h_i\}\ \langle$**send** $mktok(t_i \vee b_i)$ **to** $i-1 \quad h_i := false \quad b_i := false\rangle\ \{\mathcal{I}\}$

First, we show that execution of this statement changes tp from i to $i-1$. Since h_i holds, we conclude (using I) that $tp = i$. It follows from the definition of tp that $(\forall j \neq i+1\colon lx_j \prec lx_{i+1})$. Since h_i holds, I implies $\sigma_i^{tok} \neq \emptyset$ and $\sigma_i^{tok} = \rho_i^{tok}$. Let m be the element of σ_i^{tok} with the largest timestamp; thus, $lx_{i+1} = ts(m)$. Since $\sigma_i^{tok} = \rho_i^{tok}$, $m \in \rho_i$, so (using I) $ts(m) \preceq vt_i$, i.e., $lx_{i+1} \preceq vt_i$. Thus, by transitivity of $\prec$, $(\forall j \neq i+1\colon lx_j \prec vt_i)$. Since this statement does not affect lx_j for $j \neq i$, after execution of this statement, $(\forall j \notin \{i, i+1\}\colon lx_j \prec vt_i)$ holds. Also after execution of the statement, $lx_i = inc(vt_i, i)$. By definition of $\prec$, $vt_i \prec inc(vt_i, i)$, so by transitivity, $(\forall j \notin \{i, i+1\}\colon lx_j \prec vt_i)$ holds after execution. Since $lx_{i+1} \preceq vt_i \prec inc(vt_i, i)$, after execution, $lx_{i+1} \prec lx_i$ holds. Thus, after execution, $(\forall j \neq i\colon lx_j \prec lx_i)$ holds, so by definition of tp, $tp = i-1$.

J_1 is preserved because after execution of this statement, lx_i is larger than the timestamps of all messages previously sent by process i. To show that J_{tok} is preserved, it suffices to show $lx_{i+1} \preceq inc(vt_i, i)$, since $lx_i = inc(vt_i, i)$ after execution. Let m be the member of σ_i^{tok} with the largest timestamp (this is well-defined since h_i and I imply that $\sigma_i^{tok} \neq \emptyset$ and that the timestamps of messages in σ_i^{tok} are totally-ordered by $\prec$); thus, $lx_{i+1} = ts(m)$. Since h_i holds, we conclude using I that $\sigma_i^{tok} = \rho_i^{tok}$, so $m \in \rho_i$, which implies (using I) that $ts(m) \preceq vt_i$. By definition of $\prec$, $vt_i \prec inc(vt_i, i)$. Thus, $lx_{i+1} \preceq vt_i \prec inc(vt_i, i)$.

Next we show that J_{bas} is preserved. Fix j, k, and $m \in \chi_{j,k}$ (we have renamed the bound variable i in (13) to j). We do a case analysis on the relative values of j, k, and tp.

case $k \leqslant tp < j$. Since J_{bas} holds, $nlx_j \prec ts(m)$. If $tp \neq k$, then $k \leqslant tp < j$ is preserved by execution of this statement, so we must show $nlx_j \prec ts(m)$, which we already know to be true. Suppose $tp = k$. After execution of this statement, $\neg(k \leqslant tp < j)$, so we must show $lx_j \prec ts(m)$. We give a proof by contradiction: we suppose $\neg(lx_j \prec ts(m))$ and show $m \in \rho_k$, which contradicts the assumption $m \in \chi_{j,k}$. I implies that the timestamps generated by each process are totally ordered by $\prec$, so $ts(m) \preceq lx_j$. Since $tp = i$, J_{tok} implies $lx_j \preceq lx_{j-1} \cdots \preceq lx_{i+1}$, so $ts(m) \preceq lx_{i+1}$. Let m' be the member of σ_i^{tok} with the largest timestamp (this is well-defined since h_i and I imply that $\sigma_i^{tok} \neq \emptyset$ and that the timestamps of messages in σ_i^{tok} are totally-ordered by $\prec$); thus, $lx_{i+1} = ts(m')$, so $ts(m) \preceq ts(m')$. Since h_i holds, we conclude (using I) $\sigma_i^{tok} = \rho_i^{tok}$, so (using I) $m' \in \rho_i$, hence (again using I) $ts(m') \preceq vt_i$. Thus, $ts(m) \preceq ts(m') \preceq vt_i$, so (using I) $m \in \rho_i$. Since by assumption $i = k$, $m \in \rho_k$.

case $k < j$ **and** $\neg(k \leqslant tp < j)$. Since J_{bas} holds, $lx_j \prec ts(m)$. As in the previous case, preservation of J_{bas} is trivial if $tp \neq j$. Suppose $tp = j$. After execution of this statement, $k \leqslant tp < j$, so we must show that $nlx_j \prec ts(m)$ then holds; this follows immediately from $lx_j \prec ts(m)$ and the fact that the value of nlx_j after execution of this statement equals the value of lx_j before execution of this statement.

case $j \leqslant tp < k$. This case is analogous to the previous case.

case $j \leqslant k$ **and** $\neg(j \leqslant tp < k)$. This case is analogous to the first case.

Finally, we show that K is preserved by execution of this statement. Recall that execution of this statement changes tp from i to $i - 1$. Note that execution of this statement leaves tc unchanged. The proof that K is preserved is by case analysis on the disjunct of K that holds initially.

case K_1. We distinguish two subcases.

case $(\forall k\colon \chi_{i,k} = \emptyset)$. From the precondition of this triple, q_i holds. Since execution of this statement does not affect q_i or $\chi_{i,k}$ for all k, K_1 continues to hold after execution of this statement.

case $(\exists k\colon \chi_{i,k} \neq \emptyset)$. Since K_1 and h_{tp} hold, $(\forall k \geqslant i\colon \chi_{i,k} = \emptyset)$ does too. This, together with the assumption $(\exists k\colon \chi_{i,k} \neq \emptyset)$, implies there exists k such that $k < i$ and $\chi_{i,k} \neq \emptyset$. Let m be an element of $\chi_{i,k}$. Since $k < i$ and $tp = i$, J_{bas} implies $lx_i \prec ts(m)$, from which we conclude using J_1 that b_i holds. After execution of this statement, tc equals $t_i \vee b_i$, so K_3 then holds.

case K_2. Since $i = tp$, $K_2 = (\exists k < i\colon b_k) \vee b_i$. If the left disjunct holds, then K_2 still holds after execution of this statement. If the right disjunct holds before execution, then so does K_3 (because h_i holds and $tp = i$), so see that case.

case K_3. tc is unchanged by execution of this statement, so K_3 still holds after execution of this statement.

A.2. Proof for process 0 in isolation

The verification of process i when $i = 0$ in isolation involves the following triples, in addition to those discussed above.

T10 : $\{\mathcal{I} \wedge \neg h_0 \wedge tp \geqslant 0 \wedge (\forall j\colon \sigma_j^{tok} = \emptyset)\}$
send *mktok(false)* **to** $N - 1$ $\{\mathcal{I} \wedge \neg h_0 \wedge tp \geqslant 0\}$

First, we show that after execution of this statement, $tp = N - 1$. The precondition implies $(\forall j\colon \{m \in \bigcup_k \sigma_k^{tok} \mid sender(m) = j\} = \emptyset)$; it follows from the definition of lx_j that $lx_j = \vec{0}$ for all j. After execution of this statement, $lx_0 = inc(vt_0, 0)$. From the definition of $\prec$, $\vec{0} \prec inc(vt, 0)$ for all vector times vt. From the definition of tp, we conclude that after execution of this statement, $(\forall j \neq 0\colon lx_j \prec lx_0)$ holds, hence $tp = N - 1$.

J_1 is preserved because after execution of this statement, lx_0 is larger than the timestamps of all messages previously sent by process 0. To show that J_{tok} holds after execution of this statement, we need to show that $\vec{0} \preceq \vec{0}$ and $\vec{0} \preceq inc(vt_0, 0)$; both of these facts follow from the definition of $\prec$. To see that J_{bas} holds after execution of this statement, note that $lx_j = \vec{0}$ and (by the same reasoning) $nlx_j = 0$ for $j \neq 0$. Thus, J_{bas} holds trivially for $j \neq 0$. For $j = 0$, note that there is no process k such that $k < 0$, and recall that after execution of this statement, $tp = N - 1$. Thus, the only non-vacuous conjunct in J_{bas} is the bottom one. This conjunct holds because $nlx_0 = 0$.

The conjunct $tp \geqslant 0$ in the postcondition holds after execution because tp then equals $N - 1$, as shown above. Finally, note that $\neg h_0$ is unaffected by execution of this statement.

T11 : $\{\mathcal{I} \wedge \neg h_0 \wedge tp \geqslant 0\}$ $Init_0$ $\{\mathcal{I}\}$

Validity of this triple follows by the same reasoning as for triple T1.

T12 : $\{\mathcal{I} \wedge h_0\}$ $\langle$**send** $mktok(false)$ **to** $N - 1$ $\quad h_0 := false \quad b_0 := false\rangle$ $\{\mathcal{I}\}$

J is preserved by the same reasoning as for triple T9. We now show that execution of this statement truthifies K_1. Since h_0 holds, we conclude (using I) that $tp = 0$ holds before execution of this statement, so $\neg h_{N-1}$, because otherwise, I implies $tp = N - 1$, which contradicts $tp = 0$. By the same reasoning as for triple T9, after execution of this statement, $tp = N - 1$. Thus, K_1 holds vacuously after execution of this statement.

Finally, we discuss one proof obligation that arises when using the foregoing results to verify the proof outlines given in Fig. 3. When proving the second branch of $RELAY_0$, the following subgoal arises:

$$\mathcal{I} \wedge q_0 \wedge h_0 \wedge \neg(t_0 \vee b_0) \Rightarrow Q$$

We assume the antecedent and prove the consequent. First, we show that K_1 must hold, by showing that K_2 and K_3 do not. Since h_0 holds, we conclude (using I) that $tp = 0$. From $tp = 0$ and $\neg b_0$, we conclude that K_2 does not hold. From h_0 and $\neg(t_0 \vee b_0)$, we conclude that K_3 does not hold. Thus, assuming the antecedent holds, K_1 also holds. It is easy to show that K_1 and the antecedent together imply Q.

A.3. Interference freedom

Most of the interference freedom obligations can be discharged easily, using derived rules such as Interference Freedom for Synchronously Altered Assertions [15]. One non-trivial triple that arises in the proof of interference freedom is

$$\{K[q_j := \mathit{false}] \wedge K \wedge K[q_i := \mathit{false}]\}\ R_i\ \{K[q_j := \mathit{false}]\}$$

where $j \neq i$. By the Assignment Axiom, validity of this triple follows from

$$K[q_j := \mathit{false}] \wedge K \wedge K[q_i := \mathit{false}] \Rightarrow K[q_i := \mathit{false},\ q_j := \mathit{false}]$$

We assume the antecedent and prove the consequent. If K_2 holds, then $K_2[q_i := \mathit{false},\ q_j := \mathit{false}]$ holds, since q_i and q_j do not appear in K_2. The same reasoning applies to K_3. If neither K_2 nor K_3 hold, then $K_1[q_j := \mathit{false}] \wedge K_1 \wedge K_1[q_i := \mathit{false}]$ must hold. We show by contradiction that this implies $i \leqslant tp$. Suppose $i > tp$; then

$$\begin{aligned} K_1 = {} & q_i \wedge (\forall k\colon\ \chi_{i,k} = \emptyset) \\ & \wedge (\forall i' > tp\colon\ i' \neq i \Rightarrow q_{i'} \wedge (\forall k\colon\ \chi_{i',k} = \emptyset)) \\ & \wedge (h_{tp} \Rightarrow (\forall k \geqslant tp\colon\ \chi_{tp,k} = \emptyset)) \end{aligned}$$

so $K_1[q_i := \mathit{false}] = \mathit{false} \wedge \cdots$, so $K_1[q_i := \mathit{false}]$ does not hold, which contradicts the assumption above. Thus, $i \leqslant tp$. Analogous reasoning shows that $j \leqslant tp$. Since $i \leqslant tp$ and $j \leqslant tp$, K_1 is independent of q_i and q_j. By assumption, K_1 holds, so $K_1[q_i := \mathit{false},\ q_j := \mathit{false}]$ also holds.

References

[1] A.L. Ambler et al., Gypsy: A language for specification and implementation of verifiable programs, *ACM SIGPLAN Notices* **12**(3) (1977) 1-10.

[2] K.R. Apt, Correctness proofs of distributed termination algorithms, *ACM Trans. Programming Lang. Systems* **8**(3) (1986) 388-405.

[3] K.P. Birman, The process group approach to reliable distributed computing, *Comm. ACM* **36**(12) (1993).

[4] K. Birman, A. Schiper and P. Stephenson, Lightweight causal and atomic group multicast, *ACM Trans. Comput. Systems* **9**(3) (1991) 272-314.

[5] B. Charron-Bost, F. Mattern and G. Tel, Synchronous and asynchronous communication in distributed computations, Technical Report LITP 92-77, Institut Blaise Pascal, University of Paris 7, 1992.

[6] T. Camp, P. Kearns and M. Ahuja, Proof rules for flush channels, *IEEE Trans. Software Eng.* **19**(4) (1993) 366-378.

[7] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, Derivation of a termination detection algorithm for distributed computations, *Inform. Process. Lett.* **16** (1983) 217-219.

[8] C. Fidge, Timestamps in message-passing systems that preserve the partial ordering, in: *Proc. 11th Australian Computer Science Conference* (1988) 56-66.

[9] D. Good, R. Cohen and J. Keeton-Williams, Principles of proving concurrent programs in Gypsy, in: *Conf. Record of the Sixth Ann. ACM Symp. on Principles of Programming Languages* (1979) 42-52.

[10] D. Gries, An illustration of current ideas on the derivation of correctness proofs and correct programs, *IEEE Trans. on Software Eng.* **2** (1976) 238-244.

[11] E.P. Gribomont, From synchronous to asynchronous communication, in: C. Rattray, ed., *Specification and Verification of Concurrent Systems: Proc. of a 1988 BCS-FACS Workshop*, FACS Workshop Series, Vol. 1 (Springer, Berlin, 1990).

[12] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* **12**(10) (1969) 576-580.

[13] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Comm. ACM* **21**(7) (1978) 558-564.

[14] L. Lamport, The 'Hoare logic' of concurrent programs, *Acta Inform.* **14** (1980) 21-37.

[15] G.M. Levin and D. Gries, A proof technique for communicating sequential processes, *Acta Inform.* **15** (1981) 281-302.

[16] F. Mattern, Virtual time and global states of distributed systems, in: M. Corsnard, ed., *Proc. Internat. Workshop on Parallel and Distributed Algorithms* (North-Holland, Amsterdam, 1989) 120–131.
[17] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs I, *Acta Inform.* **6** (1976) 319–340.
[18] M. Raynal and J.-M. Helary, *Synchronization and Control of Distributed Systems and Programs* (Wiley, New York, 1990).
[19] R.D. Schlichting and F.B. Schneider, Using message passing for distributed programming: proof rules and disciplines, *ACM Trans. Programming Lang. Systems* **6**(3) (1984) 402–431.
[20] A.J.M. van Gasteren and G. Tel, Comments on "On the proof of a distributed algorithm": Always-true is not invariant, *Inform. Process. Lett.* **35** (1990) 277–279.
[21] R. van Renesse, Causal controversy at le Mont St.-Michel, *Operating Systems Review* **27**(2) (1993) 44–53.
[22] P. Verjus, On the proof of a distributed algorithm, *Inform. Process. Lett.* **25** (1987) 145–147.