Specifications and Proofs for Ensemble Layers

Jason Hickey[1], Nancy Lynch[2], Robbert van Renesse[1]

[1] Dept. of Computer Science, Cornell University ***
[2] Laboratory for Computer Science, Massachusetts Institute of Technology

Abstract. Ensemble is a widely used group communication system that supports distributed programming by providing precise guarantees for synchronization, message ordering, and message delivery. Ensemble eases the task of distributed-application programming, but as a result, ensuring the correctness of Ensemble itself is a difficult problem. In this paper we use I/O automata for formalizing, specifying, and verifying the Ensemble implementation. We focus specifically on message *total ordering*, a property that is commonly used to guarantee consistency within a process group. The systematic verification of this protocol led to the discovery of an error in the implementation.

1 Introduction

Ensemble [8, 16] is a working system for supporting group communication. In the group communication model, processes join together to form *views* that vary over time, but at any time a process belongs to exactly one view. Ensemble provides precise semantics for message delivery and ordering both within a view, and as views change. The Ensemble implementation is modular; applications acquire services by constructing layered protocol stacks. Ensemble currently provides about 50 protocol layers, and the number of useful protocols that can be constructed by composing the layers into protocol stacks numbers in the thousands.

Ensemble eases the task of distributed-application programming by supporting properties like failure detection and recovery, process migration, message ordering, and conflict resolution, through a common application interface. From one perspective, Ensemble provides a model for establishing confidence: the critical algorithms are cleanly isolated and modularized. From another perspective, the task of verifying thousands of protocols is seemingly impossible! Any verification model that we use *must* capture the modularity of Ensemble, and it must be able to provide automated assistance for module composition.

In this paper we present our experience applying I/O automata [13, 14] to Ensemble. The I/O automaton model provides a good framework

*** Support for this research was provided by DARPA contract F30602-95-1-0047 (Cornell), and DARPA contract F19628-95-C-0118, AFOSR contract F49620-97-1-0337, and NSF grants CCR-9804665 and CCR-9225124 (MIT).

for modeling Ensemble because: (a) Ensemble layers can be described formally as automata, and composition of layers corresponds to composition of automata, (b) the I/O automaton model language supports a range of specification, from abstract specifications that characterize services to operational specifications that characterize program behavior, and (c) the automata can be interpreted formally, as part of a mechanical verification we are performing with the Nuprl system [5]. We demonstrate our experience through a case study of the Ensemble *total-order* protocol, which specifies an ordering property for message delivery. It is built incrementally from *virtual synchrony*, a basic Ensemble service. We present the following contributions:

– EVS, a specification for the safety properties guaranteed by the Ensemble virtual synchrony layer.
– ETO, for the Ensemble totally ordered virtual synchrony layer.
– EVStoETO$_p$, for the local program at node p, used in Ensemble in the implementation of ETO using EVS. The original program was written in OCaml by Mark Hayden [16, 8], based on C code developed by Robbert van Renesse for the Horus system [17].
– a simulation relation showing that the composition of EVS and all the EVStoETO$_p$, for all p, implements ETO.

This document gives the specifications and summarizes the proofs for the total order case study. The full proofs are given in detail in [9], which provides the formal arguments used in the mechanical verification using the Nuprl proof development system. At the time of writing, the mechanical verification is partially complete. While we do not discuss proof automation specifically, the specifications we present were developed through a process of reverse-engineering, by hand-translating Ensemble code into a Nuprl specification, and the proofs were developed in concert with the Nuprl formalism.

The outline for the rest of the paper is as follows. In Section 2, we give a brief description of the I/O automata formalism, and in Section 3, we use it to specify the abstract Ensemble client. We specify the ETO and EVS services in Sections 4 and 5; we develop the layer specification and its verification in Section 6; and we finish with a discussion of the specific ordering properties that led to the discovery of an error in Ensemble and Horus in Section 7.

2 Notation and mathematical foundations

Sets, functions, sequences. Given a set S not containing $\perp$, the notation $S_\perp$ refers to the set $S \cup \{\perp\}$. We write $\langle\langle\rangle\rangle$ for the empty sequence. If a is a sequence, $|a|$ denotes the length of a. We also use the notation

$|a|_x$ to denote the number of elements in a that are equal to x. If a is a sequence and $1 \le i \le j \le |a|$ then $a(i)$ denotes the ith element of a and $a(i..j)$ denotes the subsequence $a(i), \ldots, a(j)$. We say that sequence s is a *prefix* of sequence t, written as $s \le t$ iff there exists i such that $s = t(1 \ldots i)$.

Views. $\mathcal{P}$ denotes the universe of all processes. $\mathcal{G}$ is a totally ordered set of identifiers used to distinguish views. Within $\mathcal{G}$, we distinguish view identifiers g_p, $p \in \mathcal{P}$, one per process p. We assume that these special view identifiers come before all other view identifiers in the given total ordering of $\mathcal{G}$. A *view* $v = \langle g, P \rangle$ consists of a view identifier g, $g \in \mathcal{G}$ and a nonempty set P, $P \in 2^{\mathcal{P}}$, of processors called "members" of the view. $\mathcal{V} = \mathcal{G} \times 2^{\mathcal{P}}$ is the set of all views. Given a view $v = \langle g, P \rangle$, the notation $v.id$ refers to the view identifier g of view v and the notation $v.set$ refers to the view membership set P of view v. We distinguish special *initial views* $v_p = \langle g_p, \{p\} \rangle$ for all $p \in \mathcal{P}$. In specifications that associate at most one view with each identifier $g \in \mathcal{G}$, we will sometimes refer to the "view" g, meaning the view with identifier g.

Messages. We denote by $\mathcal{M}$ the universe of all possible messages. When messages are placed in queues, they are often paired with processors $\mathcal{M} \times \mathcal{P}$. Given a message-processor pair $x = \langle m, p \rangle$, the notation $x.msg$ refers to the message m, and $x.proc$ refers to the processor p.

I/O automata. I/O automata provide a *reactive* model for programs that react with their environment in an ongoing manner, as described by Lynch [14]. An automaton consists of a set of *actions*, classified as *input*, *output*, or *internal*, a (possibly inifinite) set of *states*, and a set of *transitions*, which are (*state*, *action*, *state*) triples. A valid *execution* is a state-action sequence $s_1 a_1 \ldots s_i a_i s_{i+1} \ldots$ where each triple $s_i a_i s_{i+1}$ is a transition of the automaton. The I/O automata pseudocode we use in this paper describes the automaton in three parts: (1) the possible actions are described in the *signature*, (2) the state is expressed as a collection of variables and their domains, (3) the transitions are described with precondition/effect clauses for each action.

3 The client automaton C_p

The specification of the Ensemble client is shown in Figure 1. The client automaton is used to formalize restrictions on the environment in which Ensemble services exist. There is one client C_p per process $p \in \mathcal{P}$; each client represents a single process in an Ensemble application. The group membership changes over time in three distinct phases, represented by three modes.

The client is initialized in the "normal" mode, and it can communicate with other processes in the view by sending and receiving messages. When

C_p

Signature:

Input: ETO-BLOCK_p, $p \in \mathcal{P}$
$\text{ETO-NEWVIEW}(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
$\text{ETO-GPRCV}(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$

Output: ETO-BLOCK-OK_p, $p \in \mathcal{P}$
$\text{ETO-GPSND}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$

State:

$mode \in \{$ *"normal"*, *"preparing"*, *"blocked"*$\}$, initially *"normal"*

Transitions:

input $\text{ETO-NEWVIEW}(v)_p$
Eff: $mode := normal$

input ETO-BLOCK_p
Eff: $mode := preparing$

output ETO-BLOCK-OK_p
Pre: $mode = preparing$
Eff: $mode := blocked$

output $\text{ETO-GPSND}(m)_p$
Pre: $mode \neq blocked$
Eff: *none*

input $\text{ETO-GPRCV}(m)_{p,q}$
Eff: *none*

Fig. 1. The C_p specification

a new view is to be installed, Ensemble notifies the client by sending it a BLOCK message. The BLOCK message puts the client in the "preparing" mode; the client may continue to send and receive messages in the "preparing" mode. The client may respond to the BLOCK request with a BLOCK-OK message, which makes the client "blocked." The client is not allowed to send messages in the blocked mode. The transition from the "blocked" to the "normal" mode occurs when Ensemble delivers the NEWVIEW message, which installs a new view in the client with a potentially new list of view members.

4 Ensemble virtual synchrony (EVS)

Virtual Synchrony provides the semantics of group communication. The view guarantees provided by Ensemble can be summarized with the following informal properties. *EVS-self*: if process p installs view v, then $p \in v.set$. *EVS-view-order*: views are installed in ascending order of view id. *EVS-non-overlap*: for any two processes p and q that both install view v, the previous views of p and q must either be the same or be disjoint.

Failures may prevent messages from being delivered, and virtual synchrony provides the following delivery guarantees. *EVS-msg-view*: all delivered messages are delivered in the view in which they were sent. *EVS-fifo*: messages between any two processes in a view are delivered in FIFO order. *EVS-sync*: any two processes that install a view v_2, both with preceding view v_1, deliver the same messages in view v_1.

The automaton for EVS is shown in Figure 2. This automaton contains a state shared by all processes, and the external events in the signature are indexed by processes $p \in \mathcal{P}$. There is one event to match each of the

EVS

Signature:

In: $\text{EVS-GPSND}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
EVS-BLOCK-OK_p, $p \in \mathcal{P}$
Internal: $\text{EVS-CREATEVIEW}(v)$, $v \in \mathcal{V}$

Out: $\text{EVS-GPRCV}(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$
EVS-BLOCK_p. $p \in \mathcal{P}$
$\text{EVS-NEWVIEW}(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

created $\subseteq \mathcal{V}$, initially $\{v_p : p \in \mathcal{P}\}$
for each $p \in \mathcal{P}$:
  mode$[p] \in \{$ *"normal"*, *"preparing"*, *"blocked"*$\}$, initially *"normal"*
  all-viewids$[p] \subseteq \mathcal{G}$, initially $\{g_p\}$
for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  pending$[p, g] \in$ *seqof*$(\mathcal{M})$, initially $\langle\langle\rangle\rangle$
for each $p, q \in \mathcal{P}$, $g \in \mathcal{G}$:
  next$[p, q, g] \in \mathbf{N}^+$, initially 1

Derived variables:

for each $p \in \mathcal{P}$:
  all-views$[p] \subseteq \mathcal{V}$, given by $\{v \in$ *created* $: v.id \in$ *all-viewids*$[p]\}$
  current-viewid$[p] \in \mathcal{G}$, given by *max*(*all-viewids*$[p]$)
  current-view$[p] \in \mathcal{G}$, a default view $v \in$ *created* such that
    $v.id =$ *current-viewid*$[p]$
for each $g \in \mathcal{G}$, $p \in \mathcal{P}$:
  pred-viewid$[g, p] \in \mathcal{G}_\perp$, the largest viewid strictly smaller than
    g in *all-viewids*$[p]$, if $g \in$ *all-viewids*$[p]$ and any such viewid exists, else $\perp$
for each $v \in \mathcal{V}$, $p \in \mathcal{P}$:
  pred-view$[v, p] \in \mathcal{V}$, a default view $w \in$ *all-views*$[p]$ such that
    $w.id =$ *pred-viewid*$[v.id, p]$, if $v \in$ *all-views*$[p]$ and any such w exists, else $\perp$

Transitions:

output EVS-BLOCK_p
  Pre: *mode*$[p] =$ *normal*
  Eff: *mode*$[p] :=$ *preparing*
input EVS-BLOCK-OK_p
  Eff: *mode*$[p] :=$ *blocked*

output $\text{EVS-NEWVIEW}(v)_p$ choose $v1$
  Pre: *mode*$[p] =$ *blocked*
    $v1 =$ *current-view*$[p]$
    $v \in$ *created*
    $v.id > v1.id$
    $\forall q \in v.set$:
      if *pred-view*$[v, q] \neq \perp$ then
        pred-view$[v, q] = v1 \lor$ *pred-view*$[v, q].set \cap v1.set = \{\}$
      if *pred-view*$[v, q] = v1$ then
        $\forall r \in v1.set$:
          next$[r, p, v1.id] =$ *next*$[r, q, v1.id]$
  Eff: *mode* $:=$ *normal*
    all-viewids$[p] :=$ *all-viewids*$[p] \cup \{v.id\}$

internal $\text{EVS-CREATEVIEW}(v)$
  Pre: $\forall w \in$ *created* $: v.id > w.id$
  Eff: *created* $:=$ *created* $\cup \{v\}$
input $\text{EVS-GPSND}(m)_p$
  Eff: append m to *pending*$[p,$ *current-viewid*$[p]]$
output $\text{EVS-GPRCV}(m)_{q,p}$, choose g
  Pre: $g =$ *current-viewid*$[p]$
    pending$[q, g] \neq \langle\langle\rangle\rangle$
    pending$[q, g]($*next*$[q, p, g]) = m$
  Eff: *next*$[q, p, g] :=$ *next*$[q, p, g] + 1$

Fig. 2. EVS specification

client events. In addition there is a new internal action EVS-CREATEVIEW(v) that creates new views that may eventually be installed.

In the state, we keep a history for each process. The variable $mode[p]$ represents the mode of client C_p. The sequence $all\text{-}viewids[p]$ is the history of all views that have been delivered to process p. The sequence $pending[p, g]$ is the sequence of messages sent by process p in view g. The index $next[q, p, g]$ indicates the next message to be delivered to process p from process q in view g (so $pending[q, g](next[q, p, g])$ is the next message to be delivered). The view $current\text{-}view[p]$ is the last view that was delivered to the client, and $pred\text{-}view[g, p]$ is the view delivered just before view g to process p.

The transitions for EVS-BLOCK and EVS-BLOCK-OK represent state changes in the client. The transition for EVS-GPSND$(m)_p$ places the message m in the current sequence of pending messages for process p, and the transition for EVS-GPRCV$(m)_{q,p}$ takes a message from the pending queue for process q and delivers it to process p.

The EVS-NEWVIEW$(v)_p$ transition requires several properties before a new view can be delivered to the client C_p. The precondition $v.id > v1.id$ requires that the new view be larger than the current view (which ensures *EVS-view-order*). For each process $q \in \mathcal{P}$, the precondition $pred\text{-}view[v, q] = v1 \lor pred\text{-}view[v, q].set \cap v1.set = \{\}$ provides the *EVS-non-overlap* property for processes that have already installed view v ($pred\text{-}view[v, q] \neq \perp$). The precondition $next[r, p, v1.id] = next[r, q, v1.id]$ provides the *EVS-sync* property: the messages delivered from process r must be the same for all processes that have installed view v from view $v1$. These properties, together with the *EVS-fifo* property that follows from the ordering of messages in the *pending* queues, yield the informal properties claimed by the designers.

5 Ensemble total order (ETO)

The ETO service guarantees all of the properties of EVS, and also the following ordering guarantees on message delivery. *ETO-total*: Any two messages m_1 and m_2 delivered to more than one process are delivered in the same order. *ETO-causal*: Messages are *causally* ordered: if process p_2 receives a message m from process p_1, then it must have received all messages received by p_1 before m was sent.

The automaton for ETO is derived from EVS, with the differences shown in Figure 3: 1) the EVS-$\cdots$ actions of EVS are renamed with the ETO-$\cdots$ prefix, 2) the transitions for ETO-GPSND$(m)_p$ and ETO-GPRCV$(m)_{q,p}$ replace the corresponding transitions of EVS, 3) ETO-ORDER$(m, i, j)_p$ is a new action, and 4) the ETO state *adds* the state variables $queue[g]$ and $pending[p, g]$ to the state of EVS. The total order for each view $g \in \mathcal{G}$ is

ETO: changes from EVS

Signature:

In: ETO-GPSND$(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
ETO-BLOCK-OK$_p$, $p \in \mathcal{P}$
Internal: ETO-CREATEVIEW(v), $v \in \mathcal{V}$
ETO-ORDER$(mf, i, j)_p$, $mf \in \mathcal{M} \times (\mathcal{P} \to \mathbf{N}^+)$, $i, j \in \mathbf{N}^+$, $p \in \mathcal{P}$

Out: ETO-GPRCV$(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$
ETO-BLOCK$_p$. $p \in \mathcal{P}$
ETO-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

for each $g \in \mathcal{G}$:
$queue[g] \in seqof(\mathcal{P})$, initially $\langle\langle\rangle\rangle$
for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
$pending[p, g] \in seqof(\mathcal{M} \times (\mathcal{P} \to \mathbf{N}^+))$, initially $\langle\langle\rangle\rangle$

Derived variables:

$enabled[p, q, g] \in bool$, indicates when a totally ordered message can be delivered from process p to process q in view g:

$$\begin{aligned}&\exists i.\ queue[g](i) = p\\ &\quad \wedge \left|queue[g](1 \ldots i)\right|_p = next[p, q, g]\\ &\quad \wedge \forall p' \in \mathcal{P}.p' \neq p \Rightarrow \exists j < i\colon \left|queue[g](1 \ldots j)\right|_{p'} = next[p', q, g] - 1\end{aligned}$$

Transitions:

input ETO-GPSND$(m)_p$
Eff: choose $g = current\text{-}viewid[p]$
choose $f = \lambda r.next[r, p, g]$
append (m, f) to $pending[p, g]$

internal ETO-ORDER$(mf, i, j)_p$, choose g
Pre: $\left|queue[g](1 \ldots i)\right|_p = j - 1$
$\left|queue[g]\right|_p = j - 1$
$pending[p, g](j) = mf$
Eff: insert p into $queue[g]$ at i

output ETO-GPRCV$(m)_{q,p}$, choose g, f
Pre: $g = current\text{-}viewid[q]$
$pending[q, g](next[q, p, g]) = (m, f)$
$\forall r \in \mathcal{P}.next[r, p, g] \geq f(r)$
$enabled[q, p, g]$
Eff: $next[q, p, g] := next[q, p, g] + 1$

Fig. 3. The specification modifications for ETO

represented by the $queue[g]$ process sequence, where message m_i in the total order is from process $queue[g](i)$. The $queue[g]$ entries are inserted by the internal action ETO-ORDER$(m, i, j)_p$, which inserts process p into the total order $queue[g]$ at location i after all other occurrences of process p in the total order.

The message delivery ordering at process p is determined by the precondition for the EVS-GPRCV$(m)_{q,p}$. The precondition $\forall r \in \mathcal{P}.next(r, q, g) \geq f(r)$ provides causal ordering: the ETO-GPSND$(m)_p$ transition saves a causal "snapshot" of the delivery state when the message was sent, and the $\forall r \in \mathcal{P}\colon next[r, q, g] \geq f(r)$ is the causality requirement. Total ordering is determined by the *enabled* predicate: if $enabled[p, q, g]$ then there is some index i into the total order $queue[g]$ where the number of delivered messages from each process $p' \in \mathcal{P}$ is no more than the number of occurrences of p' in $queue[g](1 \ldots i)$. This condition for ordering allows message deliveries that contain gaps. For example, consider the ordering $queue[g] = \langle\langle \underline{p_1}\ \underline{p_2}\ \underline{p_1}\ \underline{p_2}\ p_1\ p_3\ \underline{p_2}\ p_3\ p_2 \rangle\rangle$, where the underlined process identifiers represent messages that have been delivered to process p. Two messages have been delivered from processes p_1 and p_2. Message deliveries from p_1 and p_3 are no longer enabled—they would violate the total order. The only possible future delivery is from process p_2.

6 The implementation algorithm (EVStoETO)

Ensemble implements services using separate protocol stacks for each process. The layer that implements total-order uses a two-phase token-based algorithm. When a view is first installed with the EVS-NEWVIEW$(v)_p$ action a token is generated by the *group leader* (the process with the smallest process identifier). Each message sent during the first phase, called the *ordered* phase, must be associated with a token. When a process has a message to send, it is required to obtain a token. If it obtains a token t_i, it sends the message with the token, and generates new token t_{i+1}. During this phase, the sent messages $(m_1, t_1), (m_2, t_2), \ldots$ can be totally ordered by their tokens.

When messages are received by the layer from EVS in the ordered phase, they are saved in a queue, called the *ordered* queue, in the order determined by their tokens. The EVStoETO$_p$ layer delivers message m_i to the client C_p only if messages $m_1, m_2, \ldots, m_{i-1}$ have been successfully received by the layer (with the EVS-GPRCV$(m)_{q,p}$ action) and delivered to the client (with the ETO-GPRCV$(m)_{p,q}$ action).

The second phase of the protocol, called the *unordered* phase, can be entered by the layer at any time. During the unordered phase, outgoing messages are sent without waiting for the token, and they are designated as "unordered." Layers that receive unordered messages place them on a queue called the *unordered* queue. Delivery of an unordered message to the client is delayed until the installation of the next view, upon which the layer sorts the contents of the unordered queue by process-identifier, and delivers the queued messages to the client before delivering the new view.

The specification for the EVStoETO layer is shown in Figures 4 and 5[1]. In this specification, tokens for messages in the *ordered* mode are represented by their number. The layer for EVStoETO uses four message types to communicate information about messages and their ordering: *Ordered*(t, m) pairs token t with message m, *Unordered*(m) designates an unordered message, *TokenReq* is used to request a token from another process, and *TokenSend*(t, p) is used to deliver token t to process p.

The signature for the EVStoETO layer includes both actions for communicating with EVS (the EVS-$\cdots$ events), and with the client (the ETO-$\cdots$

[1] This version fixes the original error in Ensemble and Horus, which differed in the implementation of the precondition for ETO-GPRCV$(m)_{q,p}$, as discussed in Section 7.

EVStoETO$_p$

Signature: define $\mathcal{M}_T \equiv$ *Ordered* (t, m), $t \in \mathbf{N}$, $m \in \mathcal{M}$
| *Unordered* (m), $m \in \mathcal{M}$
| *TokenReq*
| *TokenSend* (t, r), $t \in \mathbf{N}$, $r \in \mathcal{P}$

Input:
EVS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}_T$, $p, q \in \mathcal{P}$
EVS-BLOCK$_p$, $p \in \mathcal{P}$
EVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in \mathcal{P}$
ETO-BLOCK-OK$_p$, $p \in \mathcal{P}$
ETO-GPSND$(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$

Output:
EVS-GPSND$(t, m)_p$, $t \in \mathbf{N}$, $m \in \mathcal{M}_T$, $p \in \mathcal{P}$
EVS-BLOCK-OK$_p$, $p \in \mathcal{P}$
ETO-BLOCK$_p$, $p \in \mathcal{P}$
ETO-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
ETO-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$

Internal: EVS-UNORDERED$_p$, $p \in \mathcal{P}$

State:
blocked $\in$ *bool*, initially *false*
have-block $\in$ *bool*, initially *false*
have-block-ok $\in$ *bool*, initially *false*
have-newview $\in$ *bool*, initially *false*
holds-token $\in$ *bool*, initially *true*
token $\in \mathbf{N}$, initially 1
use-token $\in$ *bool*, initially *true*
view $\in \mathcal{V}$, initially v_p

request $\in 2^{\mathcal{P}}$, initially $\{\}$
requested $\in$ *bool*, initially *false*
pending $\in$ *seqof*$(\mathcal{M})$, initially $\langle\!\langle\rangle\!\rangle$
next $\in \mathbf{N}^+$, initially 1
order $\in \mathbf{N}$, initially 1
for each $t \in \mathbf{N}$:
  ordered$[t] \in (\mathcal{M} \times \mathcal{P})_\perp$, initially $\perp$
unordered $\in$ *seqof*$(\mathcal{M} \times \mathcal{P})$, initially $\langle\!\langle\rangle\!\rangle$

Transitions:
input EVS-GPRCV$(\mathit{Ordered}(t, m))_{q,p}$
  Eff: *ordered*$[t] := \langle m, q\rangle$
input EVS-GPRCV$(\mathit{Unordered}(m))_{q,p}$
  Eff: append $\langle m, q\rangle$ to *unordered*
input EVS-GPRCV$(\mathit{TokenReq})_{q,p}$
  Eff: *request* := *request* $\cup \{q\}$
input EVS-GPRCV$(\mathit{TokenSend}(t, r))_{q,p}$
  Eff: **if** $r = p \wedge$ *use-token* **then**
    holds-token := *true*
    token := t
input ETO-GPSND$(m)_p$
  Eff: append m to *pending*

input EVS-BLOCK$_p$
  Eff: *have-block* := *true*
input EVS-NEWVIEW$(v)_p$
  Eff: *have-newview* := *true*
    view := v
input ETO-BLOCK-OK$_p$
  Eff: *have-block-ok* := *true*
    blocked := *true*
internal ETO-UNORDERED$_p$
  Pre: *true*
  Eff: *holds-token* := *false*
    use-token := *false*

Fig. 4. State, input, and internal transitions for EVStoETO

events). In the specification, a process p is allowed to initiate unordered mode at any time with the internal event EVS-UNORDERED$_p$. The state has three parts.

The *view* part maintains information about the view state and pending views. The *blocked* flag is true iff the client is considered to be blocked. The *have-block*, *have-block-ok*, and *have-new-view* flags keep track of queued block events as they are passed between EVS and the client; for instance, *have-block* is set in the transition for EVS-BLOCK$_p$, and reset in the transition for ETO-BLOCK$_p$. The *view* field is valid if the flag *have-new-view* is set, and it contains the next view to be delivered to the client.

The next part of the state is for *token-management*. The *holds-token* flag is set iff the process is known to hold a valid token; the token is

Output transitions

output ETO-BLOCK$_p$
Pre: *have-block*
$\neg$*have-newview*
Eff: *have-block* := *false*

output EVS-BLOCK-OK$_p$
Pre: *have-block-ok*
$next = |pending| + 1$
Eff: *have-block-ok* := *false*

output EVS-GPSND$(Ordered(t, m))_p$
Pre: $pending(next) = m$
holds-token
$token = t$
Eff: $next := next + 1$
$token := t + 1$

output EVS-GPSND$(Unordered(m))_p$
Pre: $\neg$*use-token*
$pending(next) = m$
Eff: $next := next + 1$

output ETO-NEWVIEW$(v)_p$
Pre: *have-newview*
$v = view$
$ordered[order] = \bot$
$\forall i$: $unordered(i).proc \notin view.set$
Eff: *have-newview* := *false*
$token := 1$
$order := 1$
$next := 1$
$pending := \langle\langle\rangle\rangle$
for each $t \in \mathbf{N}$
$ordered[t] := \bot$
holds-token $= \neg\exists q \in v.set.q <_{\mathcal{P}} p$
$request := \{\}$
$requested := false$
$blocked := false$
use-token := *true*

output ETO-GPRCV$(m)_{q,p}$ choose i, j
Pre: $ordered[order] = \langle m, q\rangle$
$\vee$ (*have-newview*
$\wedge\, i \geq order \wedge j = 0$
$\wedge\, ordered[i] = \langle m, q\rangle$
$\wedge\, q \in view.set$
$\wedge\, \forall j \in [order \ldots i-1]$: $ordered[j] \neq \bot$
$\Rightarrow ordered[j].proc \notin view.set)$
$\vee$ (*have-newview*
$\wedge\, \forall k \geq order$: $ordered[k] \neq \bot$
$\Rightarrow ordered[k].proc \notin view.set$
$\wedge\, unordered(j) = \langle m, q\rangle \wedge i = 0$
$\wedge\, \forall k < j.\forall p' = unordered[k].proc$:
$p' \notin view.set \vee p' >_{\mathcal{P}} q$
$\wedge\, \forall k > j.\forall p' = unordered[k].proc$:
$p' \notin view.set \vee p' \geq_{\mathcal{P}} q)$
Eff: **if** $ordered[order] = \langle m, p\rangle$ **then**
$order := order + 1$
else if $j = 0$ **then**
$ordered[i] := \bot$
else
remove element j from *unordered*

output EVS-GPSND$(TokenReq)_p$
Pre: $next \leq |pending|$
use-token
$\neg$*holds-token*
$\neg$*requested*
Eff: *requested* := *true*

output EVS-GPSND$(TokenSend(t, r)_p)$
Pre: $next = |pending| + 1$
holds-token
$token = t$
$r \in request$
Eff: *holds-token* := *false*
$request := request - \{r\}$
requested := *false*

Fig. 5. Output transitions for EVStoETO

represented as a number stored in the *token* field. The *use-token* flag is true iff the layer is in the *ordered* phase of the protocol. The *request* field is a set of processes known to be requesting the token. The *requested* flag is set iff process p is actively requesting the token.

The final part of the state is for *ordering* and *queueing*. The *pending* field contains the messages sent by the client in the current view. The *next* field is the index of the next message to be sent to EVS from the pending queue. The *ordered* queue is the queue of ordered messages that have been received by the layer in the current view. The *order* field is the index of the last ordered message that was delivered to the client from the *ordered* queue. Unordered messages are stored in the *unordered* queue until the arrival of the next view.

An ordered message is sent to EVS with the EVS-GPSND$(m)_p$ action when the process has the token and a pending message; pending messages are sent unordered only after the unordered phase is initiated.

The ordering part of the protocol is implemented in the transition for ETO-GPRCV$(m)_{p,q}$. There are three cases where a message can be delivered to the client: (1) The next ordered message $\mathit{ordered}[\mathit{order}]$ has been queued. In this case, the message is delivered to the client, and the *order* field is incremented. (2) A new view is pending, there is a ordered message m from process q in the ordered queue, and q survives in the new view. The message is delivered to the client and removed from the ordered queue. (3) A new view is pending, all messages in the ordered queue belong to *failed* processes (processes that are not in the new view), and message m is the first message from a surviving process q. The message is delivered to the client, and removed from the unordered queue.

The new view is delivered to the client only after all messages from surviving process have been delivered to the client from the *ordered* and *unordered* queues. All messages from failed processes are discarded.

The layer verification is a forward simulation relation, as described in Chapter 8 of Lynch [14], showing that the *implementation*, EVS composed with all the layers EVStoETO$_p$ and clients C_p for each $p \in \mathcal{P}$, implements the *specification* ETO composed with all the clients C_p for each $p \in \mathcal{P}$. We implement the specification as the automaton S, and the implementation as automaton T. We abbreviate T.EVStoETO$_p$ with the notation L_p (for "layer" p), and T.EVS as V (for **V**irtual synchrony). The specification for the automaton S is the composition of ETO and C_p for each $p \in \mathcal{P}$.

For the implementation T, we define additional derived variables that correspond to values in the specification S, as shown in Figure 6. The $\mathit{mpending}[p, g]$ is the list of pending messages in the EVS automaton from process p in view g. The $\mathit{qcount}[p, q]$ is the number of messages from process p that have been queued for process q by the layer EVStoETO$_q$. The $\mathit{mcount}[p, q]$ is the number of messages from process p that have been *delivered* to process q by the layer EVStoETO$_q$. The $\mathit{next}[p, q]$ is the index into $\mathit{mpending}[p, g]$ of the *next* message to be delivered from process p to process q by the layer EVStoETO$_q$. The $\mathit{lpending}[p, q]$ field is the list of messages, both ordered and unordered, that are queued in the layer EVStoETO$_q$ for delivery to process q.

These variables provide the state correspondence shown in Figure 7. The proof of the simulation relation is by induction on the length of executions. We summarize the proof here.

First, we show that the V.EVS-GPSND$(m)_p$ corresponds to the action S.ETO.ETO-ORDER$(m, i, j)_p$. The index j is the number of messages $L_p.\mathit{next}$. We choose the index i as follows. If $m = \mathit{Ordered}(t, m')$ is an ordered mes-

T

Compose:

EVS
for each $p \in \mathcal{P}$:
  EVStoETO$_p$
  C_p

Hide:

EVS-GPSND$(m)_p$, $m \in \mathcal{M}_T$, $p \in \mathcal{P}$
EVS-BLOCK-OK$_p$, $p \in \mathcal{P}$
EVS-GPRCV$(m)_{p,q}$, $m \in \mathcal{M}_T$, $p, q \in \mathcal{P}$
EVS-BLOCK$_p$. $p \in \mathcal{P}$
EVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

Derived variables:

for each $p \in \mathcal{P}$:
  current-viewid$[p] \in \mathcal{G}$, given by EVS.*current-viewid*$[p]$ if $\neg$EVStoETO.*have-newview*, or EVS.*pred-viewid*[EVS.*current-viewid*$[p], p]$ otherwise. This view represents the "current view" of the EVStoETO layer.
for each $g \in \mathcal{G}$:
  oqueue$[g] \in$ *seqof*$(\mathcal{P})$, where
  oqueue$[g](i) = p$
    if there is a pending entry j where EVS.*pending*$[p, g](j) =$ *Ordered*(i, m).
    The length $|$*oqueue*$[g]|$ is the number of pending ordered messages.
  uqueue$[g] \in$ *seqof*$(\mathcal{P})$, where *uqueue*$[g](i) = p$
    if there is a pending entry j where EVS.*pending*$[p, g](j) =$ *Unordered*(m),
      and $|$*uqueue*$[g]|_p$ is the number of unordered messages in *pending*$[p, g]$,
      and *uqueue*$[g]$ is sorted by processor using the ordering $<_{\mathcal{P}}$ of EVS
  queue$[g] \in$ *seqof*$(\mathcal{P})$ defined by *queue*$[g] =$ *oqueue*$[g] +$ *uqueue*$[g]$
for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  mpending$[p, g] \in$ *seqof*$(\mathcal{M})$ defined by the the sequence of *Ordered* and *Unordered* messages in EVS.*pending*$[p, g]$
for each $p, q \in \mathcal{P}$:
  qcount$[p, q] \in \mathbf{N}$ defined by the number of messages from processor p in EVStoETO$_q$.*ordered*(EVStoETO.*order* ...) and EVStoETO$_q$.*unordered*
  mcount$[p, q] \in \mathbf{N}$ defined by the number of *Ordered* and *Unordered* messages in EVS.*pending*$[p,$ EVS.*current-viewid*$[q]](1 \ldots$ EVS.*next*$[p, q,$ EVS.*current-viewid*$[q]])$.
  next$[p, q] \in \mathbf{N}^+$ defined by *mcount*$[p, q] -$ *qcount*$[p, q]$
  lpending$[p, q] \in$ *seqof*$(\mathcal{M})$ defined by the sequence of messages from p in EVStoETO$_q$.*ordered* + EVStoETO$_q$.*unordered*

Fig. 6. Total Order Implementation

sage, then the insertion occurs at location $i = t$. If $m = Unordered(m')$ is an unordered message, then the location i is the *last* location in $T.queue[g]$ after all ordered messages, but before any occurrences of processes $p' > p$.

Next, we show that the action T.ETO-GPRCV$(m)_{q,p}$ corresponds directly to the action S.ETO-GPRCV$(m)_{q,p}$. For this part, we need to prove that each delivery T.ETO-GPRCV$(m)_{q,p}$ is both causal and enabled (with the S.ETO.*enabled*$[q, p, g]$ predicate). The ordering argument has three parts, corresponding to the precondition for L_p.ETO-GPRCV.

For ordered messages in the first clause of the precondition, the ordering conditions are straightforward. Since every message is associated with a token, and all messages are delivered in strict token order, causality and totality are trivially preserved.

$$
\begin{array}{lrl}
① & S.\text{ETO}.created & = V.created \\
② & S.C_p.mode & = T.C_p.mode \\
③ & S.\text{ETO}.mode & = T.C_p.mode \\
④ & S.\text{ETO}.all\text{-}viewids & = \begin{cases} V.all\text{-}viewids[p] - \{L_p.view\} & \text{if } hnv \\ V.all\text{-}viewids[p] & \text{if } \neg hnv \\ \quad where\ hnv = L_p.have\text{-}newview & \end{cases} \\
⑤ & S.\text{ETO}.queue[g] & = T.queue[g] \\
⑥ & S.\text{ETO}.pending[p,g] & = L_p.pending \\
⑦ & S.\text{ETO}.next[p,q,g] & = T.next[p,q] \\
 & where\ g & = S.\text{ETO}.current\text{-}viewid(p)
\end{array}
$$

Fig. 7. State relation

The proof for the second disjunct of ETO-GPRCV$(m)_{q,p}$ is more complex because of causal relationships between queued messages at the arrival of a new view. At the new view, the *ordered* queue may contain messages interspersed with gaps for messages that were never received by the layer. The only assumption that can be made about the lost messages is that they were not received by *any* process in the new view (the *EVS-sync* property). Since the causal relationships are otherwise unknown, the implementation can only deliver messages from processes that remain in the new view. As we discuss in Section 7, the original Ensemble and Horus implementations did not implement this step exactly.

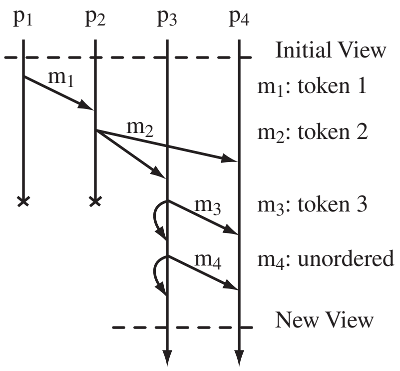
Lastly, the proof of ordering for messages in the *unordered* queue is straightforward. Since delivery of unordered messages is postponed until the next view, all unordered messages are causally unrelated. The total ordering property follows because the layers sort the messages using the ordering over $\mathcal{P}$, and causality follows because messages from failed processes are not delivered.

7 EVStoETO: discussion

The most complex part of the proof is the action for EVS-GPRCV, because three different cases have to be handled: one case for ordered messages, one for unordered, and one for ordered messages that have been received during the transition when some layers are sending ordered messages, and others are sending unordered messages. The message delivery properties of EVS do *not* guarantee that there will be no gaps in the *ordered* queue of messages, even when a new view is passed to the layer with the EVS-NEWVIEW$(v)_p$ action. This is a subtle point that involves the issue of *causal* ordering of messages.

We can illustrate the problem with a scenario involving four processes, all initially in the same view. In this scenario, message m_1 is delivered to process p_2, which immediately delivers it to the client. The client sends a new message m_2 which is received by processes p_3 and p_4, and then processes p_1 and p_2 fail.

Message m_2 is causally related to message m_1, but m_2, m_3, and m_4 are causally unrelated because clients C_3 and C_4 do not receive any messages until the arrival of the new view due to the gap in the *ordered* queue left for message m_1. Because of failures, there is no way to recover message m_1. Which messages should be delivered?



The implementations addressed this problem in two different ways. The Ensemble implementation discarded all pending *ordered* messages at the arrival of the new view, and Horus implementation delivered them all. Ensemble would discard message m_3 but deliver m_4, violating the *EVS-fifo* property, and Horus would deliver message m_2 without delivering m_1, violating *ETO-causal*. An implementation with the *EVS-fifo* and *ETO-causal* properties would deliver, at most, messages m_3 and m_4.

When we first started working on the verification problem, the first step was to derive the specifications from the Ensemble ML code, which gave us the specification EVStoETO$_p$ shown in Figure 5 *without* the middle precondition for ETO-GPRCV$(m)_{q,p}$. It became clear as we were doing the simulation proof that the simulation step for ETO-GPRCV$(m)_{q,p}$ would fail: there were some message deliveries that would not be allowed by the specification of total order ETO. The solution seemed to be either to strengthen the properties of EVS or strengthen the precondition for ETO-GPRCV.

When we spoke with the developers about this problem, we found a line of reasoning common to both implementations: if EVS were to preserve *causal* ordering of messages, the implementations would work correctly. However, causal ordering is *not* provided by EVS for efficiency reasons; applications that need causal ordering add an additional protocol layer to implement the property. The code was corrected by implementing the additional precondition and effect for ETO-GPRCV$(m)_{q,p}$. The changes to the implementation code were minimal, and both implementations have since been corrected.

8 Related work

Birman and Joseph presented one of the earliest accounts of virtual synchrony [4] in 1987. Since that time many group membership and communication specifications have appeared. An article published in 1995 [1] points out that many attempts of these have been unsatisfactory. Several new specifications have appeared that do not suffer from the shortcomings in [1], such as [15, 7, 2, 3]. A specification of protocol layers and their composition appeared in [17]. Automata are used for specifying distributed systems in [10, 6]. In [11], protocol layers for point-to-point messaging are formally specified and composed using TLA [12].

References

1. Emmanuelle Anceaume, Bernadette Charron-Bost, Pascale Minet, and Sam Toueg. On the formal specification of group membership services. Technical Report TR 95-1534, Cornell University Computer Science Department, August 1995.
2. Ozalp Babaoglu, Renzo Davoli, L. Giachini, and G. Baker. System support for partition-aware network applications. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, May 1998.
3. Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, January 1997.
4. Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc 11th Symposium on Operating Systems Principles (SOSP)*, pages 123–138, November 1987.
5. R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System.* Prentice–Hall, 1986.
6. Alan Fekete. Formal models of communications services: A case study. *IEEE Computer*, 26(8):37–47, August 1993.
7. Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using partitionable group communication service. In *Proc.16thAnnual ACM Symposium on Principles of Dist. Comp.*, pages 52–62, 1997.
8. Mark G. Hayden. *The Ensemble System.* PhD thesis, Dept. of Computer Science, Cornell University, January 1997.
9. Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. Technical Report forthcoming, MIT and Cornell University, 1998. available at `http:// www.cs.cornell.edu/jyh/papers/HLR98.ps`.
10. Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.
11. David A. Karr. *Protocol Composition on Horus.* PhD thesis, Dept. of Computer Science, Cornell University, December 1996.
12. Leslie Lamport. Introduction to TLA. Technical Report 1994-001, DIGITAL SRC, Palo Alto, CA, 1994.
13. Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands*, 2(3):219–246, September 1989. Also Tech. Memo MIT/LCS/TM-373.
14. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
15. Gil Neiger. A new look at membership services. In *Proc.15thAnnual ACM Symposium on Principles of Dist. Comp.*, pages 331–340, May 1996.
16. Robbert Van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software–Practice and Experience*, 29(9):963–979, July 1998.
17. Robbert Van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A Framework for Protocol Composition in Horus. In *Proc. 14th Annual ACM Symposium on Principles of Dist. Comp.*, pages 80–89, Ottawa, Ontario, August 1995. ACM SIGOPS-SIGACT.