An Object-Oriented Approach to Verifying Group Communication Systems

Mark Bickford[2] and Jason Hickey[1]

[1] Cornell University***
[2] Odyssey Research Associates

Abstract. Group communication system assist the development of fault-tolerant distributed algorithms by providing precise guarantees on message ordering, delivery, and synchronization. Ensemble is a widely used group communication system that is highly modular and configurable. Formally verifying Ensemble is a formidable task, but it has wide-ranging benefits, from formal assistance in the design of new distributed applications, to ensuring the reliability of critical distributed algorithms for all applications that use Ensemble.

In this paper, we present a verification framework that we are using the verify Ensemble in the Nuprl proof development system. The framework is based on I/O automata, which are ideal for the verification in some respects: they they specify modular components that range from concrete protocol code to abstract services. But traditional I/O automata do not allow re-use of formal theorems as automata are composed. We present a new type-theoretic basis for I/O automata that preserves safety properties during composition using an object-oriented methodology.

1 Introduction

This paper presents two efforts: the implementation of a formal framework for verifying modular systems, and the application of the framework to verify Ensemble. Ensemble is a highly modular group communication system that eases the task of distributed programming by providing configurable implementations of critical distributed algorithms like failure detection and recovery, while giving precise guarantees for message ordering and delivery. This is ideal from the perspective of reliability: the critical distributed algorithms are cleanly isolated and identified. Formal verification of these protocols applies to all applications that use Ensemble.

The Ensemble architecture is designed around layered protocol stacks, composed from a collection of protocol layers that provide a roughly orthogonal set of primitives. Each layer is about 300 lines of code in Objective Caml [17]. Currently there are about 50 different layers that can be composed to implement thousand of useful protocols. Clearly, any feasible formal framework must capitalize on modularity and re-use.

*** Support for this research was provided by DARPA grant F30602-95-1-0047.

In this paper, we present a formal framework for specifying and verifying modular systems like Ensemble based on the I/O automata of Lynch and Tuttle [15, 14]. I/O automata provide an ideal specification language in some respects. They can specify modular components that range from the concrete OCaml layer code to abstract distributed services. However, the automata defined by Lynch and Tuttle do not, in general, allow formal theorems to be re-used as protocols are modified. For example, a FIFO protocol may have the informal specification, "all messages are delivered in the order in which they were sent." When a total ordering layer is added to the protocol, it adds the additional property, "all processes receive messages in the same order." In the standard formalization of I/O automata, the TOTAL protocol is different from the FIFO protocol, and the FIFO property must be re-proved after the total-ordering property is added.

We introduce a new type-theoretic foundation for I/O automata that preserves safety properties as automata are composed. The definition of composition is based on the *intersection* of automata: if A and B are automata, and A has safety property P (written $A \models \mathit{always}\, P$), then $A \cap B \models \mathit{always}\, P$. In fact, the sentence $(A \models \mathit{always}\, P) \Rightarrow (A \cap B \models \mathit{always}\, P)$ is a formal *theorem* in our framework. The intersection of automata corresponds to inheritance in object-oriented languages. Our paradigm advocates the construction of specifications from a library of small, easily-understood components, using compositional intersection to construct service specifications.

We embed the automata in the Nuprl type theory, which provides a rich, expressive logic and programming environment. There are several advantages of doing this. Since the logic is expressive, we can develop highly readable and compact specifications. Furthermore, we can completely justify our formalism; the properties and meta-principles of automata are Nuprl theorems. Nuprl also provides support for proof automation so that the tools and methodology for reasoning can be defined and implemented.[1] It should not be construed that higher-order logic is *necessary*—rather our work in higher-order logic provides a firm foundation on which to base reasoning systems for use in simpler, less expressive logics.

In the remainder of the paper, we cover the formal framework in more detail. In Section 2 we develop the framework by adding intersections to "standard" I/O automata and giving the type theoretic interpretation. In Section 3 we present the specification of virtual synchrony, a major service provided by group communication systems. Our work shares many of the features of conjoined TLA specifications described by Abadi and Lamport [1]; we discuss related work in Section 4.

2 I/O automata

General I/O automata are instances of *labeled transition systems*. An automaton consists of a set of states Q; a set of labels A, called *actions*, that label each of

[1] The formal system and reasoning tool presented in this paper are available at the Nuprl WWW site at `www.cs.cornell.edu`.

the transitions of the automaton; a set of *initial* states i; and a set of allowable transitions t. In type theory, an automaton is described by the following type:

$$\begin{aligned} Auto \equiv \quad & Q\colon Type \\ \times\ & A\colon Type \\ \times\ & i\colon Q \to \mathbb{P} \\ \times\ & t\colon (Q \times A \times Q) \to \mathbb{P} \end{aligned} \tag{1}$$

The initial states i are a subset of the possible states Q, defined using a predicate $Q \to \mathbb{P}$, where $\mathbb{P}$ is the set of propositions (for simplicity, this can be thought of as a Boolean value). The transition definition is similar. Note that the states and actions may be types with an infinite number of elements.

I/O automata are often defined using stylized pseudo-code. The *actions* correspond to "significant" external events of the system being specified. For example, a simple specification of a FIFO protocol is shown below. The FIFO actions are SEND(m) and RECV(m), where m ranges over some message type $\mathcal{M}$. The automaton contains two queues that save the messages *sent* and *received*. When a message is sent, with the transition labeled SEND(m), the message is added to the *sent* list of messages. Messages are received with the transition labeled RECV(m). A message m can be received when there are more messages in the *sent* queue than the *received* queue and the next message in *sent* is m (this is a *precondition* of the transition); the effect is to add the message to the *received* queue.

FIFO	
Actions:	SEND(m), RECV(m), for $m \in \mathcal{M}$
State:	*sent* $\in \mathcal{M}$ *List*, initially empty,
	received $\in \mathcal{M}$ *List*, initially empty
SEND(m)	
Eff: append m to *sent*	
RECV(m)	
Pre: $\lvert received \rvert < \lvert sent \rvert$	
$\quad sent[\lvert received \rvert - 1] = m$	
Eff: append m to *received*	

In general, an automaton defines a set of *executions* $q_0, a_0, q_1, a_1, \ldots$, where each triple q_i, a_i, q_{i+1} is a valid transition of the automaton. It also defines a set of *traces*, which are the actions of the valid executions. If A and B are automata, and the traces of A are a subset of the traces of B, we say that A *implements* B. An *invariant* is a predicate that is true of all states in all executions of an automaton. The FIFO automaton defines that invariant that the *received* queue is always a prefix of the *sent* queue: FIFO $\models$ *always received* $\leq$ *sent*.

So far, this presentation of I/O automata adheres to the original form[2] of Lynch and Tuttle. Now suppose we wish to define a form of composition that preserves safety properties, which include state invariants and properties of individual traces of the automaton. The obvious choice for composition is the

[2] Lynch and Tuttle also labels actions as being *input*, *output*, or *internal* to give a little more semantic content, but we find this labeling unnecessary.

intersection of their traces. If M and M' are automata, their intersection has states that are states both of M and M', and traces that belong both to M and M'. If $M = (Q, A, i, t)$ and $M' = (Q', A', i', t')$, we define their intersection as follows:

$$\begin{array}{rl} M \cap M' \equiv (& Q \cap Q', \\ & A \cap A', \\ & \lambda s.(i(s) \wedge i'(s)), \\ & \lambda q_1, a, q_2.t(q_1, a, q_2) \wedge t'(q_1, a, q_2)). \end{array} \tag{2}$$

With this definition, we have the following theorem, proved formally in Nuprl.

Theorem 1. *For any automata M and M, and state predicate P, if $M \models$ always P, then $M \cap M' \models$ always P.*

The proof is straightforward, since any execution of $M \cap M'$ is also an execution of M. In the degenerate case, if the state spaces Q and Q' do not have any elements in common, the intersection $M \cap M'$ is the empty automaton; it has no executions or traces. In our framework, we address this problem by including "all" state variables and actions in every automata, but leaving almost all of them unspecified, as shown in the following section.

2.1 Stylized automata

The pseudo-code suggests a *style* of automata definition that can be used to guide their construction. In the pseudo-code, the actions have a *label* (like SEND or RECV), and a *value* (like $m \in \mathcal{M}$) that is associated with the label. Given a type of labels *Label*, the set of stylized actions can be defined as the dependent sum $\mathit{Action} \equiv \Sigma l\colon \mathit{Label}.F_A(l)$ where $F_A \in \mathit{Label} \to \mathit{Type}$ is a type-function that defines the type of the value associated with each label. For example, the actions of the FIFO automaton are defined by the following Σ-type:

$$\Sigma l\colon \mathit{Label}.(\textbf{if } l = \text{“SEND”} \vee l = \text{“RECV”} \textbf{ then } \mathcal{M} \textbf{ else } \mathit{Top}).$$

If the action label is SEND or RECV, the action *value* belongs to the type of messages $\mathcal{M}$. Otherwise, the data part is totally unspecified (the type *Top* is the type containing all terms, defined as the degenerate intersection $\bigcap_{x \in \mathit{Void}} .x$).

The stylized states are defined similarly. A state has a set of named variables (we can use the *Label* type), and each variable has a value. The most natural type is the record type. For example, the FIFO state can be represented by the record $\{\mathit{sent}\colon \mathcal{M}\ \mathit{List}; \mathit{received}\colon \mathcal{M}\ \mathit{List}\}$. Records are dependent-function spaces that map variable names to values, and the state type is defined as $\mathit{State} = \Pi l\colon \mathit{Label}.F_S(l)$ for some type-function $F_S \in \mathit{Label} \to \mathit{Type}$. The FIFO state has the following definition:

$$\Pi l\colon \mathit{Label}.(\textbf{if } l = \text{“}\mathit{sent}\text{”} \vee l = \text{“}\mathit{received}\text{”} \textbf{ then } \mathcal{M}\ \mathit{List} \textbf{ else } \mathit{Top}).$$

Again, the state is *unspecified* on labels that are not in $\{\text{“}\mathit{sent}\text{”}, \text{“}\mathit{received}\text{”}\}$. This results in the main type-theoretic principle we use to avoid degeneracy: record *intersection* corresponds to record *concatenation*.

$$\{l_1\colon T_1; l_2\colon T_{2a}\} \cap \{l_2\colon T_{2b}; l_3\colon T_3\} \equiv \{l_1\colon T_1; l_2\colon T_{2a} \cap T_{2b}; l_3\colon T_3\}.$$

The intersection property for the Σ type also has the concatenation property. When two automata are intersected, their state variables and action signatures are concatenated. This simple feature captures the object-oriented nature of our framework: the intersection of automata corresponds to *inheritance* in object-oriented languages.

2.2 Type-theoretic interpretation

The pseudo-code can be represented in the type-theory by separating the pseudo-code into the four parts of an automaton. The actions define a Σ type, the state variables define a record (a Π type), and the initial values of the state variables specify the initial conditions. The transition definitions define a transition relation: for an automata with state type Q, the transition definition $L(x)$ `Pre` $P(q,x)$ `Eff:` $expr(q,x)$ for $x \in T$ defines a conjunctive clause in the transition definition as follows:

$$\forall x \in T.(a = L(x) \Rightarrow P(q_1, x) \wedge (q_2 = expr(q_1, x) \in Q)).$$

That is, the transition (q_1, a, q_2) is in the transition relation only if, whenever the action a is $L(x)$ for some $x \in T$, the precondition P holds on the initial state q_1, and the result of the effect is equal to the final state q_2 *on the variables specified in the state* Q. Other variables are unconstrained. The `Eff:` clause of a transition definition is optional; if it is omitted, the equality condition on q_2 is omitted in the type-theoretic interpretation, allowing the state to change arbitrarily. The formal definition of the FIFO automaton is the following program:

$$\begin{array}{l}
Q = \Pi l \in \mathit{Label}.(\textbf{if } l = \text{“}\mathit{sent}\text{”} \vee l = \text{“}\mathit{received}\text{”} \textbf{ then } \mathcal{M}\ \mathit{List} \textbf{ else } \mathit{Top}),\\
A = \Sigma l \in \mathit{Label}.(\textbf{if } l = \text{“SEND”} \vee l = \text{“RECV”} \textbf{ then } \mathcal{M} \textbf{ else } \mathit{Top}),\\
i = \lambda q.(q.\mathit{sent} = [] \wedge q.\mathit{received} = []),\\
\begin{array}{lll}
t = \lambda q_1, a, q_2. & \forall m \in \mathcal{M}.(a = \mathrm{SEND}(m) \Rightarrow & (q_2.\mathit{sent} = m \text{ appended to } q_1.\mathit{sent}\\
 & & \wedge\ q_2.\mathit{received} = q_1.\mathit{received}))\\
 & \wedge\ \forall m \in \mathcal{M}.(a = \mathrm{RECV}(m) \Rightarrow & (|q_1.\mathit{received}| < |q_1.\mathit{sent}|\\
 & & \wedge\ q_1.\mathit{sent}[|q_1.\mathit{received}| - 1] = m\\
 & & \wedge\ q_2.\mathit{received} = m \text{ appended to } q_1.\mathit{received}\\
 & & \wedge\ q_2.\mathit{sent} = q_1.\mathit{sent}))
\end{array}
\end{array}$$

2.3 Using the automata

Now, how do we use these automata? To give an example, let's modify the FIFO automaton so that no more than 100 messages are ever delivered. We construct a new automaton with the same state variable *received* used in the FIFO automaton, and we include a precondition on the $\mathrm{RECV}(m)$ action.

FINITE_FIFO	
Actions:	$\mathrm{RECV}(m)$ for $m \in \mathcal{M}$
States:	*received* $\in \mathcal{M}$ *List*, initially []
$\mathrm{RECV}(m)$	
Pre: $\lvert received \rvert < 100$	

The new automaton is not too useful by itself: it allows arbitrary state changes in the effect for RECV(m). When combined with the FIFO automaton, however, the new machine FIFO $\cap$ FINITE_FIFO has the *same* transitions and states of the FIFO automaton, except for the transition RECV(m), which has the following definition:

RECV(m) `Pre:` $\lvert received \rvert < \lvert sent \rvert$ $sent[\lvert received \rvert - 1] = m$ $\lvert received \rvert < 100$ `Eff: append` m `to` *received*

The FIFO invariant $received \leq sent$ remains true in the new automaton, but now we can also prove that the received queue never contains more than 100 messages.

2.4 Operations on automata

There are some drawbacks to the automata we have just presented. First, every state machine shares the same state space—in essence, all state variables are global. We use this feature during composition (like the FINITE_FIFO example), but there is no notion of "public" or "private" variables, where private variables do not become shared during composition. Second, the action space is global as well, so it is not possible to re-use an automaton in multiple specifications with different action namings.

Both of these problems can be addressed by introducing *renaming* operations. We use the following informal notation, given an arbitrary renaming function $R \in Label \to Label$ on *labels*, and an automaton M:

$$\begin{array}{l} M\text{, renaming actions } l(x) \text{ to } R(l)(x) \\ M\text{, renaming variables } v \text{ to } R(v). \end{array}$$

When actions are renamed, the action type and the transition definitions must be modified. Consider the following stylized machine M:

$$M = (Q = \Pi l \in Label.F_S(l), A = \Sigma l \in Label.F_A(l), i, t).$$

The action renaming "M*, renaming actions* $L(x)$ *to* $R(L)(x)$" defines the renamed machine M' shown below.

$$\begin{array}{rl} M' = (& Q' = \Pi l \in Label.F_S(l), \\ & A' = \Sigma l \in Label. \left(\bigcap_{\{l' \in Label \mid R(l')=l\}} .F_A(l') \right) \\ & i' = i, \\ & t' = \lambda q_1, l(x), q_2. \textbf{if} \quad \exists l' \in Label.(R(l') = l) \textbf{ then} \\ & \qquad\qquad\qquad\qquad t(q_1, l'(x), q_2) \\ & \qquad\qquad\qquad \textbf{else} \\ & \qquad\qquad\qquad\qquad q_1 = q_2 \in Q' \end{array}$$

Since R is an arbitrary renaming function, it may map several labels to a single value. The action type associated with label l in A' is the *intersection* of all the action types $F_A(l')$ for any label l' in the inverse image $R^{-1}(l)$. If $R^{-1}(l)$ is nonempty, the transition definition allows *any* of the transitions $l'(x)$ for any $l' \in R^{-1}(l)$. Otherwise, the value type on label l in A' degenerates to *Top*, and a transition is allowed only if it leaves the state unchanged. From this definition, we get the following formal meta-theorem for safety properties P:

Theorem 2. *For any automaton M, renaming function R, and state predicate P, if $M \models$ always P then $(M$, renaming actions $l(x)$ to $R(l)(x)) \models$ always P.*

The state renaming operation is similar. The states, initial state predicate, and transition relation must be modified. The following machine defines the renamed machine "$M'' = m$, *renaming variables* v *to* $R(v)$":

$$\begin{aligned} M'' = (\ Q'' &= \Pi l \in Label. \left(\bigcap_{\{l' \in Label \mid R(l')=l\}} .F_S(l')\right), \\ A'' &= \Sigma l \in Label. F_A(l), \\ i'' &= (i \circ R) \\ t'' &= \lambda q_1, a, q_2.t(q_1 \circ R, a, q_2 \circ R) \end{aligned}$$

This definition induces a renaming operation on state predicates P, providing the following formal meta-theorem:

Theorem 3. *For any automaton M, renaming function R, and state predicate P, if $M \models$ always P, then $(M$, renaming variables v to $R(v)) \models$ always$(\lambda q.P(q \circ R))$.*

3 Verifying Ensemble Virtual Synchrony

To demonstrate the use of our new version of I/O automata, we present the specification of Ensemble Virtual Synchrony (EVS). In the EVS model, processes join together to form *views*, which are sets of processes that vary over time as processes join and leave. Views are thought of informally as active multicast domains; when a process fails, or when the network becomes partitioned, a view may be split into several smaller views. When processes are created, or when the network heals, multiple views may merge into one. Each process has its own version of the view it is in. When a process replaces its view $view_1$ with a new view $view_2$, we say the process *installs* view $view_2$. Virtual synchrony provides the following informal properties on *views* and messages:

EVSsingle At any time a process belongs to exactly one view.
EVSself If a process installs a view, it belongs to the view.
EVSvieworder At any process, views are installed in increasing order of view identifier.
EVSnonoverlap If two processes install the same view, their previous views are either the same or they are disjoint.
EVSviewmessage All delivered messages are delivered in the view in which they were sent.
EVSfifo Messages between any two processes in a view are delivered in FIFO order.
EVSsync Any two process that install a view v_2, both with preceding view v_1, deliver the same messages in view v_1.

We can categorize these properties in three parts: message delivery (EVSfifo, EVSviewmessage), view properties (EVSsingle, EVSself, EVSvieworder, EVSnonoverlap), and message/view properties (EVSsync).

We can specify the message delivery properties using the FIFO automaton. Let $VID \subseteq Label$ be the type of views, and let $PID \subseteq Label$ be the type of processes. We implement the *Label* type as a recursive type including names and closed under pairing (which we used for subscripting by process and view), to get the following automaton:

$$\text{EVS\_FIFO} = \bigcap_{v \in VID} \bigcap_{p,q \in PID} .\,(\text{FIFO renaming} \quad \begin{array}{l} \text{action SEND}(m) \text{ to EVS-SEND}_{p,v}(m) \\ \text{action RECV}(m) \text{ to EVS-RECV}_{q,p,v}(m) \\ \text{variable } \mathit{sent} \text{ to } \mathit{sent}_{p,v} \\ \text{variable } \mathit{received} \text{ to } \mathit{received}_{q,p,v}) \end{array} \tag{3}$$

The EVSfifo property follows trivially from the FIFO automaton. The EVSviewmessage property follows because FIFO channels are only established within views (they do not cross views).

For the *view* properties, we introduce a new action EVS-NEWVIEW$_p(v)$, which delivers the view $v \in VID \times PID\ Set$ to process $p \in PID$. We refer to the view identifier as $v.id$ and the set of processes that belong to the view as $v.set$. We also need to introduce a history variable $\mathit{all\text{-}viewids}_p$, for each process $p \in PID$, containing the set of views ever delivered to process p. Several derived variables are defined in terms of $\mathit{all\text{-}views}_p$:

- $\mathit{current\text{-}view}_p$ is the view in $\mathit{all\text{-}views}_p$ with the largest identifier.
- $\mathit{pred\text{-}view}_{p,v}$ is the view in $\mathit{all\text{-}views}_p$ with identifier strictly smaller than $v.id$ if such a view exists, otherwise $\perp$, where $\perp \notin VID$ is a constant.

Given these definitions the view part of EVS is shown below.

EVS_VIEW
State: for each $p \in PID$:
$\mathit{all\text{-}viewids}_p \in \mathit{View\ Set}$, initially $\{v_p\}$
EVS-NEWVIEW$_p(v)$
Pre: let $v' = \mathit{current\text{-}view}_p$ in
$v.id > v'.id$ ①
$p \in v.set$ ②
$\forall q \in v.set.$ ③
if $\mathit{pred\text{-}view}_{q,v} \neq \perp$ then
$\mathit{pred\text{-}view}_{q,v} = v' \vee \mathit{pred\text{-}view}_{q,v}.set \cap v'.set = \{\}$
Eff: $\mathit{all\text{-}views}_p = \mathit{all\text{-}views}_p \cup \{v\}$

The precondition for EVS-NEWVIEW$(v)_p$ has several parts, one for each of the view properties. Part ① guarantees that views are delivered in ascending order (EVSvieworder), and the definition of $\mathit{current\text{-}view}_p$ defines exactly one current view per process (EVSsingle). Part ② ensures that a process belongs to all its view (EVSself). Part ③ sets up the condition on previous views: if another process q has installed view v, then it either has the same previous view, or its previous view was disjoint from v (EVSnonoverlap).

The final part of EVS relates views and message ordering. For this automaton, we need to express the relation between the FIFO and VIEW automata, and we include state variables $received_{q,p,v}$ and $all\text{-}views_p$. The VIEW_MSG automaton is shown below.

VIEW_MSG	
State:	for $p, q \in PID$, $v \in View$:
	$received_{p,q,v} \in \mathcal{M}\ List$
	for $p \in PID$:
	$all\text{-}views_p \in View\ Set$
EVS-NEWVIEW$_p(v)$	
Pre: let $v' = current\text{-}view_p$ in	
$\forall q \in v.set$:	
$\forall r \in v'.set$. if $pred\text{-}view_{q,v} = v'$ then	
$received_{r,p,v'.id} = received_{r,q,v'.id}$	

The VIEW_MSG automaton introduces a new precondition for delivering a view: each process q with the same preceeding view v' receives exactly the same messages from each process $r \in v'.set$ (they EVSsync property).

These three automata specify the properties of EVS individually. The final step is combine them into the complete specification of EVS.

$$EVS \equiv \text{GROUP\_FIFO} \cap \text{VIEW} \cap \text{VIEW\_MSG} \tag{4}$$

This construction identifies the $received_{p,q,v}$ variables of the GROUP_FIFO and VIEW_MSG automata, and the $all\text{-}views_p$ variables of the VIEW and VIEW_MSG automata. The properties of EVS are the conjunction of the properties of the three parts, forming a complete specification of EVS.

4 Related Work

Abadi and Lamport [1] have explored composition in TLA using *assume-guarantee* specifications. We describe our systems with automata, rather than a temporal *logic* like TLA, because we can use a single language to represent both programs and specifications. However, our formalism shares the same problem space with TLA. In fact, our safety theorem $(A \models always\ P) \Rightarrow (A \cap B \models always\ P)$ is currently too weak: while the intersection $A \cap B$ has the safety properties of both A and B, additional safety properties may hold because of interference between the two automata. It is likely that we can use the assume-guarantee style of reasoning to extend our framework with a more complete result.

Although our approaches differ at a high level—TLA is a temporal *logic* and we deal directly with automata, both approaches describe the same problem space.

Automata are used for specifying distributed systems in [11, 6]. In [12], protocol layers for point-to-point messaging are formally specified and composed using TLA [13].

I/O automata are defined by Lynch and Tuttle [14], and are discussed further in Lynch [15]. The Nuprl proof development system includes both a logic and a mechanism for reasoning. An early version of the system is described in Constable et. al. [5]; more recent descriptions can be found in Jackson's thesis [10]. Records are a central part of our formalization; a more complete description of the type theory of records can be found in Hickey [9].

Birman and Joseph presented one of the earliest accounts of virtual synchrony [4]; more recent versions can be found in [16, 7, 2, 3]. A more complete description of the verification of EVS can be found in [8].

References

1. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Toplas*, 17(3), May 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. Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In *TACAS '99*, March 1999.
9. Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page at Williams College.
10. Paul Bernard Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, January 1995.
11. Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. on Prog. Lang. and Systems*, 16(2):259–303, March 1994.
12. David A. Karr. *Protocol Composition on Horus.* PhD thesis, Dept. of Computer Science, Cornell University, December 1996.
13. Leslie Lamport. Introduction to TLA. Technical Report 1994-001, DIGITAL SRC, Palo Alto, CA, 1994.
14. 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.
15. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
16. Gil Neiger. A new look at membership services. In *Proc.15thAnnual ACM Symposium on Principles of Dist. Comp.*, pages 331–340, May 1996.
17. Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.