Proving Hybrid Protocols Correct

Mark Bickford, Christoph Kreitz, Robbert van Renesse, Xiaoming Liu

Department of Computer Science, Cornell University, Ithaca, NY, U.S.A.
{markb,kreitz,rvr,xliu}@cs.cornell.edu

Abstract. We describe a generic switching protocol for the construction of hybrid protocols and prove it correct with the NUPRL proof development system. For this purpose we introduce the concept of *meta-properties* and use them to formally characterize communication properties that can be preserved by switching. We also identify *switching invariants* that an implementation of the switching protocol must satisfy in order to work correctly.

1 Introduction

Formal methods tools have greatly influenced our ability to increase the reliability of software and hardware systems by revealing errors and clarifying critical concepts. Tools such as extended type checkers, model checkers [9] and theorem provers [2, 15, 23, 25] have been used to detect subtle errors in prototype code and to clarify critical concepts in the design of hardware and software systems. System falsification is already an established technique for finding errors in the early stages of the development of hardware circuits and the impact of formal methods has become larger the earlier they are employed in the design process.

An engagement of formal methods at an early stage of the design depends on the ability of the formal language to naturally and compactly express the ideas underlying the system. When it is possible to precisely define the assumptions and goals that drive the system design, then a theorem prover can be used as a design assistant that helps the designers explore in detail ideas for overcoming problems or clarifying goals. This formal design process can proceed at a reasonable pace, if the theorem prover is supported by a sufficient knowledge base of basic facts about systems concepts that the design team uses in its discussions.

The NUPRL Logical Programming Environment (LPE) [10, 3] is a framework for the development of formalized mathematical knowledge that is well suited to support such a formal design of software systems. It provides an expressive formal language and a substantial body of formal knowledge that was accumulated in increasingly large applications, such as verifications of a logic synthesis tool [1] and of the SCI cache coherency protocol [13] as well as the verification and optimization of communication protocols [17, 12, 18].

We have used the NUPRL LPE and its database of thousands of definitions, theorems and examples for the formal design of an adaptive network protocol for the ENSEMBLE group communication system [28, 11, 19]. The protocol is realized as a hybrid protocol that *switches* between specialized protocols. Its design was centered around a characterization of communication properties that can be

P. Jackson, ed., *14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, LNAI xxx, pp. 1–16,

preserved by switching. This led to a study of *meta-properties*, i.e. properties of properties, as a means for classifying those properties. It also led to the characterization of a *switch-invariant* that an implementation of the switch has to satisfy to preserve those properties.

In this paper we show how to formally prove such hybrid protocols correct. In Section 2 we describe the basic architecture of hybrid protocols that are based on protocol switching. We then discuss the concept of meta-properties and use it to characterize *switchable* properties, i.e. communication properties that can be preserved by switching (Section 3). In Section 4 we give a formal account of communication properties and meta-properties as a basis for the verification of hybrid protocols with the NUPRL system. In Section 5 we develop the switch-invariant and formally prove that switchable properties are preserved whenever the implementation of a switching protocol satisfies this invariant.

2 Protocol Switching

Networking properties such as total order or recovery from message loss can be realized by many different protocols. These protocols offer the same functionality but are optimized for different environments or applications. *Hybrid protocols* can be used to combine the advantages of various protocols, but designing them correctly is difficult, because they require a distributed migration between different approaches to implementing the desired properties.

The ENSEMBLE system [28, 11] provides a mechanism for *switching* between different protocols at run-time. So far, however, it was not clear how to guarantee that the result was actually *correct*, i.e. under what circumstances a switch would actually preserve the properties of the individual protocols.

Our new approach to switching is to design a *generic switching protocol* (SP) that would serve as a wrapper for a set of protocols with the same functionality. This switching protocol shall interact with the application in a transparent fashion, that is, the application cannot tell easily that it is running on the SP rather than on one of the underlying protocols, even as the SP switches between protocols. The kinds of uses we envision include the following:

- *Performance.* By using the best protocol for a particular network and application behavior, performance can always be optimal.
- *On-line Upgrading.* Protocol switching can be used to upgrade network protocols or fix minor bugs at run-time without having to restart applications.
- *Security.* System managers will be able to increase security at run-time, for example when an intrusion detection system notices unusual behavior.

In a protocol layering architecture like the one used in ENSEMBLE the switching protocol will reside on top of the individual protocols to be switched, coupled by a multiplexer below them, as illustrated in Figure 1.

The basic idea of the switching protocol is to operate in one of two modes. In *normal mode* it simply forwards messages from the application to the current protocol and vice versa. When there is a request to switch to a different

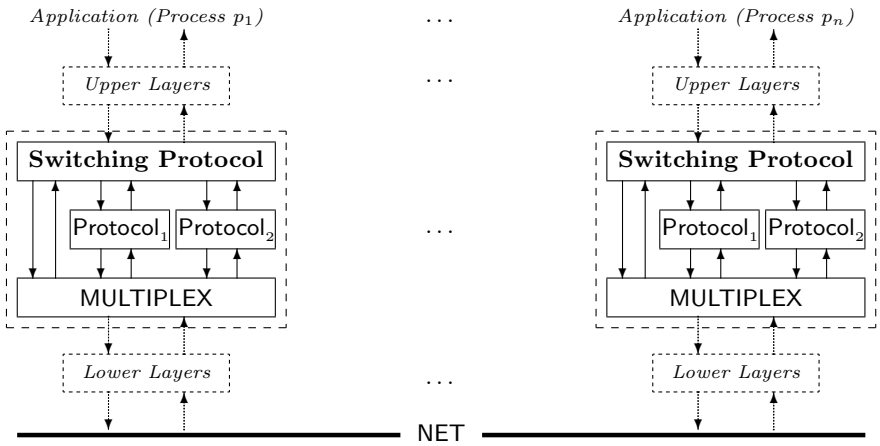


Fig. 1. Generic switching protocol for a protocol layering architecture

protocol, the SP goes into *switching mode*, during which the switching layer at each process will deliver all messages that were sent under the old protocol while buffering messages sent under the new one. The SP will return to normal mode as soon as all messages for the old protocol have been delivered.

The above coarse description, which trivially generalizes to switching between an arbitrary number of protocols, served as the starting point for proving the correctness of the resulting hybrid protocol. The verification proceeded in two phases. We first classified communication properties that are *switchable*, i.e. have the potential to be preserved under switching, and then derived a *switching invariant* that a switching protocol must satisfy to preserve switchable properties. The implementation of the SP (see [19]) was strongly influenced by the latter.

3 Meta-Properties: Classifying Switchable Properties

The main purpose of a switching protocol is to dynamically reconfigure a communication protocol without noticeable effects for the application. This is not possible for switching between arbitrary, functionally equivalent protocols, as the current protocol does not have information about messages that were sent before the switch took place but may need this information to accomplish its task.

A notion of correctness for switching protocols can thus only be expressed relatively to the communication properties that it shall preserve. Because of the broad range of applications there are many interesting properties that one may want a communication system to satisfy. Table 1 lists some examples.

Usually, these properties are expressed as properties of global (systemwide) or local (for one process) traces of *Send*/*Deliver* events. Reliability, for instance, means that for every *Send* event in the global trace there are corresponding *Deliver* events for all receivers.

Rather than studying the effects of the switching protocol on each communication property separately, we have developed a *characterization* of the ones

Reliability:	Every message that is sent is delivered to all receivers
Integrity:	Messages cannot be forged; they are sent by trusted processes
Confidentiality:	Non-trusted processes cannot see messages from trusted ones
Total Order:	Processes that deliver the same messages deliver them in the same order
No Replay:	A message body can be delivered at most once to a process

Table 1. Examples of commonly used communication properties

that are preserved by protocol switching. For this purpose, we introduce the concept of *meta-properties*, i.e. properties of properties, that classify communication properties into those that are preserved by certain communication protocol architectures and those that are not. In this section we will give an intuitive description of such meta-properties (see [19] for details). Their formalization as used in the verification of the switching protocol will be described in section 4.2.

Our verification efforts showed that four meta-properties are important for characterizing the effects of layered communication systems.

Safety. Safety [4] is probably the best-known meta-property. Safety means that a property does not depend on how far the communication has progressed: if the property holds for a trace, then it also holds for every prefix of that trace. Total order, for instance, is safe since taking events of the end off a trace cannot reorder message delivery. Reliability, however, is not safe since chopping off a suffix containing a *Deliver* event can make a reliable trace unreliable.

Asynchrony. Any global ordering that a protocol implements on events can get lost due to delays in the send and deliver streams through the protocol layers above it. Only properties that are asynchronous, i.e. do not depend on the relative order of events of different processes, are preserved under the effects of layering. Total order is asynchronous as well, as it does not require an absolute order of delivery events at different processes.

Delayable. Another effect of layered communication is local: at any process, *Send* events are delayed on the way down, and *Deliver* events are delayed on the way up. A property that survives these delays is called *delayable*. Total order is delayable, since delays do not change the order of *Deliver* event. This meta-property is similar to delay-insensitivity in asynchronous circuits.

Send Enabled. A protocol that implements a property for the layer above typically does not restrict when the layer above sends messages. We call a property *Send Enabled* if it is preserved by appending new *Send* events to traces. Total order is obviously send enabled. *Send Enabled* and *Delayable* are related, as both are concerned with being unable to control when the application sends messages.

These four meta-properties are sufficient for properties to survive the effects of delay in any layered environment. Since the switching protocol is based on a layered architecture, these meta-properties will be important for a property to be preserved by the switching protocol. However, two additional meta-properties are necessary to describe the specific effects of switching.

Memoryless. When we switch between protocols, the current protocol may not see part of the history of events that were handled by another protocol. It thus has to be able to work as if these events never happened. A property is *memoryless* if we can remove all events pertaining to a particular message from a trace without violating the property. That is, whether such a message was ever sent or delivered is no longer of importance. This does not imply, however, that a protocol that implements the property has to be *stateless* and must forget about the message. Total order is memoryless, since it only places conditions on events that actually take place, but its implementations are certainly not stateless.

Composable. Protocol switching causes the traces of several protocols to be glued together. Since we expect the resulting trace to satisfy the same properties as the individual traces, these properties must be *composable* in the sense that if they hold for any two traces that have no messages in common, then they also must hold for the concatenation. Total order is composable, because the concatenation of traces does not change the order of events in either trace.

Using the NUPRL system [3] we have shown that these six meta-properties are sufficient for a communication property to be preserved by the switching protocol: if such a property holds for the traces of the two protocols below the switching protocol, then it also holds for the resulting trace above the switch. In the following sections we will show how this is formally proven.

4 Formalization

A formal verification of the switching protocol with a formal proof system has to be based on an appropriate formalization of the underlying concepts in the language of the proof system. The formal language of the NUPRL proof development system [10, 3], an extension of Martin-Löf's intuitionistic Type Theory [22], already provides formalizations of the fundamental concepts of mathematics, data types, and programming.

NUPRL supports conservative language extensions by user-defined concepts via *abstractions* and *display forms*. An abstraction of the form

$$\mathit{operator\text{-}id}(\ \mathit{parameters}\ ) \equiv \mathit{expression\ with\ parameters}$$

defines a new, possibly parameterized term in terms of already existing type-theoretical expressions. Display forms can be used to change the textual representation of this term on the screen or within formal printed documents almost arbitrarily. In particular they can be used to suppress the presentation of implicit assumptions and thus ease the comprehensibility of formal text.

The NUPRL proof development system supports interactive and tactic-based reasoning in a sequent-based formalism, decision procedures, an evaluation mechanism for programs, and an extendable library of verified knowledge from various domains. A *formal documentation* mechanism supports the automated creation of "informal documents" from the formal objects. The technical report [5], containing a complete formal account of the work described in the following sections, for instance, was created entirely from within NUPRL.

4.1 A Formal Model of Communication

To support a formal verification of communication protocols, we have developed a formal model of distributed communication systems and their properties. Our model formalizes notions for the specification of distributed algorithms introduced by Lynch [20] and concepts used for the implementation of reliable network systems [6], particularly of ENSEMBLE and its predecessors [7, 27, 11].

Messages, Events, and Traces Processes multicast *messages* that contain a body, a sender, and a unique identifier. We will consider two types of *events*. A *Send*(m) event models that some process p has multicast a message m. A *Deliver*(p,m) event models that process p has delivered message m. A *trace* is an ordered sequence of *Send* and *Deliver* events without duplicate *Send* events.

In order to be able to reason formally about messages, events and traces, we introduce two classes `MessageStruct` and `EventStruct` of formal *message structures* and *event structures*, respectively.

A message structure $M \in$ `MessageStruct` provides a carrier $|M|$ and three functions, $\mathtt{content}_M$, $\mathtt{sender}_M$, and $\mathtt{uid}_M$, which compute the body, sender, and identifier of a message $m \in M$. Two messages m_1 and m_2 are considered equal, denoted by $m_1 =_M m_2$ if they have the same content, sender, and identifier.

Similarly, an event structure $E \in$ `EventStruct` provides a carrier type $|E|$, a message structure $\mathtt{MS}_E$, and three functions, $\mathtt{is\text{-}send}_E$, $\mathtt{loc}_E$, and $\mathtt{msg}_E$, where $\mathtt{is\text{-}send}_E(e)$ is `true` when the event $e \in |E|$ is a *Send* event (otherwise it is a *Deliver* event); $\mathtt{loc}_E(e)$, the *location* of the event e, is the identifier of the process that sends or receives e; and $\mathtt{msg}_E(e)$ is the message $m \in \mathtt{MS}_E$ contained in the event e. Using the latter we define a binary relation, $e_1 =^m_E e_2$, which holds if the messages contained in the events e_1 and e_2 are equal wrt. $=_M$. For example, e_1 and e_2 might be *Deliver* events of the same message m at two different locations.

Given an event structure E, a trace is just a list of events of type $|E|$. The data type of traces over E is thus defined as

$$\mathtt{Trace}_E \equiv |E|\ \mathtt{List}$$

All the usual list operations like length $|tr|$, selecting the i-th element $tr[i]$, concatenation $tr_1 @ tr_2$, prefix relation $tr_1 \sqsubseteq tr_2$, and filtering elements that satisfy a property P, $[e \in tr\,|\,P]$, apply to traces as well.

For process identifiers we introduce a (recursively defined) type `PID` that contains tokens and integers and is closed under pairing. A similar type, called `Label`, will be later be used to tag events processed by different protocols.

Properties of Traces A *trace property* is a predicate on traces that describes certain behaviors of communication. We formalize trace properties as propositions on traces, i.e. as functions from $\mathtt{Trace}_E$ to the type $\mathbb{P}$ of logical propositions.

$$\mathtt{TraceProperty}_E \equiv \mathtt{Trace}_E \to \mathbb{P}$$

All traces that we consider must satisfy at least three basic properties. Every de-

livered message must have been sent before (*causality*), no message is sent twice, and no message is delivered twice (*replayed*) to the same process. These assumptions are made implicitly in the implementation of communication systems but need to be made explicit in a formal account.

$$\begin{aligned}
&\mathtt{Causal}_E(tr)\\
&\quad\equiv\ \forall \mathtt{i}{<}|tr|.\exists \mathtt{j}{<}|tr|.\ \mathtt{j}{\le}\mathtt{i}\ \wedge\ \mathtt{is\text{-}send}_E(tr[\mathtt{j}])\ \wedge\ tr[\mathtt{j}] =^m_E tr[\mathtt{i}]\\
&\mathtt{No\text{-}dup\text{-}send}_E(tr)\\
&\quad\equiv\ \forall \mathtt{i},\mathtt{j}{<}|tr|.\ (\mathtt{is\text{-}send}_E(tr[\mathtt{i}]) \wedge \mathtt{is\text{-}send}_E(tr[\mathtt{j}]) \wedge tr[\mathtt{j}] =^m_E tr[\mathtt{i}])\\
&\qquad\qquad \Rightarrow\ \mathtt{i}=\mathtt{j}\\
&\mathtt{No\text{-}replay}_E(tr)\\
&\quad\equiv\ \forall \mathtt{i},\mathtt{j}{<}|tr|.\ (\ \neg\mathtt{is\text{-}send}_E(tr[\mathtt{i}])\ \wedge\ \neg\mathtt{is\text{-}send}_E(tr[\mathtt{j}])\\
&\qquad\qquad \wedge\ tr[\mathtt{j}] =^m_E tr[\mathtt{i}]\qquad \wedge\ \mathtt{loc}_E(tr[\mathtt{i}]) = \mathtt{loc}_E(tr[\mathtt{j}])\\
&\qquad\qquad)\ \ \Rightarrow\ \ \mathtt{i}=\mathtt{j}
\end{aligned}$$

The properties *reliability* (for multicasting), *integrity*, *confidentiality*, and *total order* (c.f. Section 3) can be formalized as follows.

$$\begin{aligned}
&\mathtt{Reliable}_E(tr)\\
&\quad\equiv\ \forall \mathtt{e}\in tr.\ \mathtt{is\text{-}send}_E(\mathtt{e})\ \Rightarrow\\
&\qquad \forall \mathtt{p}{:}\mathtt{PID}.\ \exists \mathtt{e}_1\in tr.\ \neg\mathtt{is\text{-}send}_E(\mathtt{e}_1)\ \wedge\ \mathtt{e} =^m_E \mathtt{e}_1\ \wedge\ \mathtt{loc}_E(\mathtt{e}_1)=\mathtt{p}\\
&\mathtt{Integrity}_E(tr)\\
&\quad\equiv\ \forall \mathtt{e}\in tr.\ (\neg\mathtt{is\text{-}send}_E(\mathtt{e}) \wedge \mathtt{trusted}(\mathtt{loc}_E(\mathtt{e})))\ \Rightarrow\\
&\qquad \forall \mathtt{e}_1\in tr.\ (\mathtt{is\text{-}send}_E(\mathtt{e}_1)\ \wedge\ \mathtt{e} =^m_E \mathtt{e}_1)\ \Rightarrow\ \mathtt{trusted}(\mathtt{loc}_E(\mathtt{e}_1))\\
&\mathtt{Confidential}_E(tr)\\
&\quad\equiv\ \forall \mathtt{e}\in tr.\ (\neg\mathtt{is\text{-}send}_E(\mathtt{e}) \wedge \neg\mathtt{trusted}(\mathtt{loc}_E(\mathtt{e})))\ \Rightarrow\\
&\qquad \neg(\exists \mathtt{e}_1\in tr.\ \mathtt{is\text{-}send}_E(\mathtt{e}_1)\ \wedge\ \mathtt{e} =^m_E \mathtt{e}_1\ \wedge\ \mathtt{trusted}(\mathtt{loc}_E(\mathtt{e}_1)))\\
&\mathtt{TotalOrder}_E(tr)\\
&\quad\equiv\ \forall \mathtt{p},\mathtt{q}{:}\mathtt{PID}.\ tr{\downarrow}\mathtt{p} \,\big\lfloor\, tr{\downarrow}\mathtt{q} = tr{\downarrow}\mathtt{q} \,\big\lfloor\, tr{\downarrow}\mathtt{p}
\end{aligned}$$

where $\mathtt{trusted}(p)$ characterizes trusted processes, $tr{\downarrow}p$ is the trace tr delivered at process p (the projection of all $Deliver(p,m)$ events from trace tr), and $tr_1 \,\big\lfloor\, tr_2$ denotes the restriction of tr_1 to events whose messages also occur in tr_2.

$$\begin{aligned}
tr{\downarrow}p\ &\equiv\ [\,\mathtt{e}\in tr \mid \neg\mathtt{is\text{-}send}_E(\mathtt{e}) \wedge \mathtt{loc}_E(\mathtt{e}){=}p\,]\\
tr_1 \,\big\lfloor\, tr_2\ &\equiv\ [\,\mathtt{e}_1\in tr_1 \mid \exists \mathtt{e}_2\in tr_2.\ \mathtt{e}_1 =^m_E \mathtt{e}_2\,]
\end{aligned}$$

In the following investigations we also need a notion of *refinement* on trace properties, which is defined as follows.

$$P \rhd Q\ \equiv\ \forall \mathtt{tr}{:}\mathtt{Trace}_E.\ P(\mathtt{tr})\ \Rightarrow\ \mathtt{Q}(\mathtt{tr})$$

4.2 Meta-Properties

Meta-properties are predicates on properties that are used to classify which properties are preserved by a protocol layer. In principle, any predicate on properties could be meta-property. But the meta-properties that we are specifically inter-

ested in are the ones that describe how passing events through a protocol layer affects the properties of the traces above and below that layer. We say that a reflexive and transitive relation R on traces *preserves* a property P if, whenever two traces tr_1 and tr_2 are related by R, and P holds for trace tr_1, then it also holds for tr_2. A similar definition is also given for ternary relations.

$$R \texttt{ preserves } P \equiv \forall \texttt{tr}_u,\texttt{tr}_l{:}\texttt{Trace}_E.\ (P(\texttt{tr}_l) \wedge \texttt{tr}_u\ R\ \texttt{tr}_l) \Rightarrow P(\texttt{tr}_u)$$

$$R \texttt{ preserves}_3 P \equiv \forall \texttt{tr}_u,\texttt{tr}_1,\texttt{tr}_2{:}\texttt{Trace}_E.\ (P(\texttt{tr}_1) \wedge P(\texttt{tr}_2) \wedge R(\texttt{tr}_u,\texttt{tr}_1,\texttt{tr}_2)) \Rightarrow P(\texttt{tr}_u)$$

Preservation by a (binary or ternary) relation R is thus a predicate on properties, i.e. a meta-property. Note that preservation automatically makes the effects of the relation R transitive, even if R is not. Below, we will formalize six such relations that describe the meta-properties discussed in Section 3.

Safety. Safety means that if the property holds for a trace, then it is also satisfied for every prefix of that trace. The corresponding relation R_{safety} specifies that the trace above the protocol (tr_u) is a prefix of the one below (tr_l).

$$tr_u\ \texttt{R\_safety}_E\ tr_l \equiv tr_u \sqsubseteq tr_l$$

Asynchrony. A property is asynchronous if it does not depend on the relative order of events at different processes. The corresponding relation R_{asynch} specifies that two traces are related if they can be formed by swapping adjacent events that belong to different processes. Events belonging to the same process may not be swapped.

$$tr_u\ \texttt{R\_async}_E\ tr_l \equiv tr_u\ \texttt{swap-adjacent}_{[\texttt{loc}_E(\texttt{e}) \neq \texttt{loc}_E(\texttt{e}')]}\ tr_l$$

where $tr_1\ \texttt{swap-adjacent}_{[c(e;e')]}\ tr_2$ denotes that tr_1 can be transformed into tr_2 by swapping adjacent events e and e' of tr_1 that satisfy the condition $c(e;e')$.

Delayable. A property is *delayable* if it survives delays of *Send* events on the way down and of *Deliver* events on the way up. The corresponding relation $R_{delayable}$ specifies that adjacent *Send* and *Deliver* events in the lower trace may be swapped in the upper. Events of the same kind or containing the same message may not be swapped.

$$tr_u\ \texttt{R\_delayable}_E\ tr_l \equiv tr_u\ \texttt{swap-adjacent}_{[\texttt{e} \neq^m_E \texttt{e}' \wedge \texttt{is-send}_E(\texttt{e}) \neq \texttt{is-send}_E(\texttt{e}')]}\ tr_l$$

Send Enabled. A property is *Send Enabled* if it is preserved by appending *Send* events to traces. The corresponding relation $R_{send\text{-}enabled}$ specifies that the upper trace is formed by adding *Send* events to the end of the lower trace.

$$tr_u\ \texttt{R\_send-enabled}_E\ tr_l \equiv \exists \texttt{e}{:}|E|.\ \texttt{is-send}_E(\texttt{e}) \wedge tr_u = tr_l\texttt{@[e]}$$

Memoryless. A property is *memoryless* if we can remove all events pertaining to a particular message from a trace without violating the property. The corresponding relation $R_{memoryless}$ defines that the upper trace can be formed from the one below by removing *all* events related to certain messages.

$$tr_u \ \texttt{R\_memoryless}_E \ tr_l \quad \equiv \quad \exists \texttt{e:|}E\texttt{|}.\ tr_u = [\,\texttt{e}_1 \in tr_l \mid \texttt{e} \neq^m_E \texttt{e}_1\,]$$

Composable. A property is *composable* if it is preserved when two traces that have no messages in common are concatenated. The corresponding relation $R_{composable}$ is ternary, as it characterizes the upper trace tr as concatenation of two lower traces without common messages.

$$\texttt{R\_composable}_E(tr_u, tr_1, tr_2) \quad \equiv \quad tr_u = tr_1 @ tr_2 \ \wedge \ \forall \texttt{e}_1 \in tr_1.\forall \texttt{e}_2 \in tr_2.\ \texttt{e}_1 \neq^m_E \texttt{e}_2$$

Switchable Properties. The above collection of meta-properties and its formalization is the result of a complex formal analysis of the switching protocol. The formal verification process with the NUPRL proof development system [3] required us to make many assumptions explicit that are usually implicitly present in an informal analysis of communication protocols. This is reflected in the definition of switchable communication properties. A trace property P is *switchable* if it requires the trace to be meaningful and satisfies all of the six meta-properties.

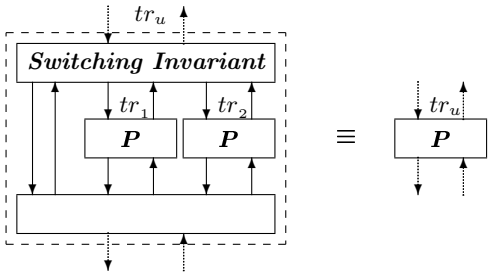
$$\begin{array}{lllll}
\texttt{switchable}_E(P) & \equiv & P & \triangleright & \texttt{Causal}_E \\
& \wedge & P & \triangleright & \texttt{No-replay}_E \\
& \wedge & \texttt{R_safety}_E & \texttt{preserves} & P \\
& \wedge & \texttt{R_async}_E & \texttt{preserves} & P \\
& \wedge & \texttt{R_delayable}_E & \texttt{preserves} & P \\
& \wedge & \texttt{R_send-enabled}_E & \texttt{preserves} & P \\
& \wedge & \texttt{R_memoryless}_E & \texttt{preserves} & P \\
& \wedge & \texttt{R_composable}_E & \texttt{preserves}_3 & P
\end{array}$$

In the following section we will show that switchable communcation properties are in fact preserved by the switching protocol.

5 Verification of Hybrid Protocols

In the previous sections we have given an abstract and formal characterization of communication protocols whose properties can be preserved by switching. In a similar way we will now develop an abstract characterization of the switching protocol in terms of a *switching invariant*. We will then prove that every implementation of a switching protocol that satisfies the switching invariant is guaranteed to preserve switchable properties. In this paper, we will focus on the highlights of this proof and the underlying formal theory. A complete account of this theory, which has been developed entirely within the NUPRL proof development system [3], can be found in the technical report that has been created automatically from the formal theory [5].

In order to prove the switching protocol to work correctly for a switchable property P we have to show that if P holds for the traces tr_1 and tr_2 of the two protocols below the switching protocol, then P also holds for the trace tr_u above the switch. That is, an application cannot tell easily that it is running a hybrid protocol with a switch instead of one of the individual protocols.



The presence of the switch has two effect on the traces. First, the two traces tr_1 and tr_2 will be merged in some way, and second, the order of some events in the merged trace may be modified due to the effects of layering.

To investigate these effects separately we introduce a *virtual middle trace* tr_m that consists of the events of tr_1 and tr_2. We will study what *local switch invariants* such a trace must satisfy to guarantee that a property, which holds on its subtraces, also hold for the whole trace. We will then link tr_m to tr_1 and tr_2 by merging and to tr_u by introducing global and local delays and additional *Send* events and derive a *global switch invariant*, which models the basic architecture of the switching protocol described in Section 2 and guarantees its correctness.

To be able to identify the origin of events in a merged trace we define a class `TaggedEventStruct` of *tagged event structures* as subtype of `EventStruct`. A tagged event structure $TE \in$ `TaggedEventStruct` provides the same components as any element of `EventStruct` but an additional function $\texttt{tag}_{TE}$ that computes the label $tg \in$ `Label` of an event $e \in |TE|$. By $\texttt{TaggedEventStruct}_E$ we denote the subclass of tagged event structures whose components as event structure are identical to those of E. Traces over tagged events are defined as before, but every event of such a trace tr is associated with a tag as well. This enables us to define the subtrace of tr that consists of all events with a given tag tg as

$$tr|_{tg} \quad \equiv \quad \texttt{[e} \in tr \texttt{ | tag}_{TE}\texttt{(e)=}tg\texttt{]}$$

Note that the notion $tr|_{tg}$ contains an implicit index TE, whose display is suppressed to simplify the notation.

We will need the following property. It characterizes tagged traces in which events with the same message have the same tag.

$$\texttt{Tag-by-msg}_{TE}(tr)$$
$$\equiv \ \forall \texttt{i,j<}|tr|\texttt{. tr[i]} =^m_{TE} \texttt{tr[j]} \ \Rightarrow \ \texttt{tag}_{TE}\texttt{(tr[i])} = \texttt{tag}_{TE}\texttt{(tr[j])}$$

5.1 The Local Switching Invariant

A local switch invariant I on a trace tr shall guarantee that a switchable property P holds for tr whenever P holds for all subtraces $tr|_{\mathtt{tg}}$. If this is the case we say that I *fuses* P:

$$I\ \mathtt{fuses}\ P\ \equiv\ \forall \mathtt{tr{:}Trace}_E.\, I(\mathtt{tr})\ \Rightarrow\ (\forall \mathtt{tg{:}Label}.\, P(\mathtt{tr}|_{\mathtt{tg}}))\Rightarrow P(\mathtt{tr})$$

From the description of the switching protocol we know that if two messages are sent using different protocols, then the second message will be buffered at a location until the first one has been delivered. In other words, if two *Send* events have different tags, then at any location, the first message must have been delivered before the second. This requirement is represented by the following invariant.

$$\begin{aligned}&\mathtt{switch\_inv}_{TE}(tr)\\ &\equiv\ \forall \mathtt{i,j,k<|}tr\mathtt{|}.\,(\mathtt{i<j}\ \wedge\ \mathtt{is\text{-}send}_{TE}(tr\mathtt{[i]})\ \wedge\ \mathtt{is\text{-}send}_{TE}(tr\mathtt{[j]})\\ &\qquad\qquad\wedge\ \mathtt{tag}_{TE}(tr\mathtt{[i]})\neq\mathtt{tag}_{TE}(tr\mathtt{[j]})\ \ \wedge\ tr\mathtt{[j]}\downarrow_{TE} tr\mathtt{[k]}\ )\\ &\qquad\Rightarrow\ \exists \mathtt{k'<k.}\ \mathtt{loc}_{TE}(tr\mathtt{[k']})=\mathtt{loc}_{TE}(tr\mathtt{[k]})\ \wedge\ tr\mathtt{[i]}\downarrow_{TE} tr\mathtt{[k']}\end{aligned}$$

where $e\downarrow_{TE} tr\mathtt{[}k\mathtt{]}$ denotes that a message related to the event e is delivered at time k in tr:

$$e\downarrow_{TE} tr\mathtt{[}k\mathtt{]}\ \equiv\ \ e=^m_{TE} tr\mathtt{[k]}\ \wedge\ \neg\mathtt{is\text{-}send}_{TE}(tr\mathtt{[k]})$$

$\mathtt{switch\_inv}_{TE}$ is sufficient to fuse switchable properties, provided the trace describes a meaningful communication, i.e. does not contain duplicate send events.

Theorem 1.

$$\begin{aligned}&\vdash\ \forall \mathtt{TE{:}TaggedEventStruct.}\ \forall \mathtt{P{:}TraceProperty}_E.\ \mathtt{switchable}_{TE}\mathtt{(P)}\\ &\quad\Rightarrow\ \mathtt{(No\text{-}dup\text{-}send}_{TE}\ \wedge\ \mathtt{switch\_inv}_{TE}\mathtt{)\ fuses\ P}\end{aligned}$$

Proof. Theorem 1 is the result of a series of steps that refine the invariants for fusing certain classes of properties until we arrive at the fusion condition for the class of switchable predicates.

1. We begin by proving an invariant for the preservation of all properties P that are memoryless, composable, and safe, denoted by the meta-property $\mathtt{MCS}_{TE}(P)$. A sufficient condition is *single-tag-decomposability*, meaning that any nonempty trace can be decomposed into two traces with no messages in common, such that all events of the (non-empty) second trace have the same tag.

$$\begin{aligned}&\mathtt{single\text{-}tag\text{-}decomposable}_{TE}(tr)\\ &\equiv\ \ tr\neq\mathtt{[]}\ \ \Rightarrow\ \ \exists \mathtt{tr_1,tr_2{:}Trace}_{TE}.\ tr=\mathtt{tr_1@tr_2}\ \wedge\ \mathtt{tr_2}\neq\mathtt{[]}\\ &\qquad\qquad\qquad\qquad\wedge\ \forall \mathtt{e_1}\in\mathtt{tr_1}.\forall \mathtt{e_2}\in\mathtt{tr_2}.\ \mathtt{e_1}\neq^m_E \mathtt{e_2}\\ &\qquad\qquad\qquad\qquad\wedge\ \exists \mathtt{tg{:}Label.}\ \forall \mathtt{e_2}\in\mathtt{tr_2.}\ \mathtt{tag}_{TE}(\mathtt{e_2})=\mathtt{tg}\end{aligned}$$

   By induction over the length of a trace we prove that any single-tag-decomposable safety property is a fusion condition for any MCS property.

$$\begin{aligned}&\vdash\ \forall \mathtt{TE{:}TaggedEventStruct.}\ \forall \mathtt{P,I{:}TraceProperty}_{TE}.\ \mathtt{MCS}_{TE}\mathtt{(P)}\ \ \Rightarrow\\ &\quad\mathtt{R\_safety}_{TE}\ \mathtt{preserves\ I}\ \ \wedge\ \ \mathtt{I}\ \triangleright\ \mathtt{single\text{-}tag\text{-}decomposable}_{TE}\\ &\quad\Rightarrow\ \ \mathtt{I\ fuses\ P}\end{aligned}$$

2. In the next step we refine single-tag-decomposability to a more constructive condition that we call *switch decomposability*. It says that it must be possible to characterize timing of some send events in a trace (i.e. their indices) by some decidable predicate Q that satisfies a number of closure conditions.

$$\begin{aligned}&\texttt{switch-decomposable}_{TE}(tr)\\ &\equiv\ tr = [] \ \vee\ \exists \texttt{Q}{:}\mathbb{N}\to\mathbb{P}.\ \texttt{decidable(Q)} \ \wedge\ \exists \texttt{i}{<}|tr|.\ \texttt{Q(i)}\\ &\qquad\wedge\ \forall \texttt{i}{<}|tr|.\ \texttt{Q(i)} \ \Rightarrow\ \texttt{is-send}_{TE}(tr[\texttt{i}])\\ &\qquad\wedge\ \forall \texttt{i,j}{<}|tr|.\ \texttt{Q(i)} \wedge \texttt{Q(j)} \ \Rightarrow\ \texttt{tag}_{TE}(tr[\texttt{i}]) = \texttt{tag}_{TE}(tr[\texttt{j}])\\ &\qquad\wedge\ \forall \texttt{i}{<}|tr|.\ \texttt{Q(i)} \ \Rightarrow\ \forall \texttt{i}{\le}\texttt{j}{<}|tr|.\ \texttt{Cl}_{\texttt{Q}}\texttt{(j)}\end{aligned}$$

where $\texttt{Cl}_Q(j)$, the *message closure* of Q at time j is defined as

$$\texttt{Cl}_Q(i) \ \equiv\ \exists \texttt{k}{<}|tr|.\ Q\texttt{(k)} \ \wedge\ tr[\texttt{k}] =^m_E tr[i]$$

By partitioning the trace into those events that satisfy $\texttt{Cl}_Q(i)$ and those that don't we can prove that switch-decomposability refines single-tag-decomposability.

$$\begin{aligned}&\vdash\ \forall \texttt{TE:TaggedEventStruct}\\ &\quad(\texttt{switch-decomposable}_{TE} \ \wedge\ \texttt{Tag-by-msg}_{TE} \ \wedge\ \texttt{Causal}_{TE} \ \wedge\ \texttt{No-dup-send}_{TE})\\ &\quad\triangleright\ \texttt{single-tag-decomposable}_{TE}\end{aligned}$$

3. In the third step we show that a certain strengthening of $\texttt{switch\_inv}_{TE}$ refines switch-decomposability. For this purpose we have to find a predicate Q that satisfies the five conditions for switch-decomposability.
If the trace satisfies $\texttt{switch\_inv}_{TE}$ and causality, then it must contain a *Send* event. We define Q to hold on all indices that are related to the index `ls` of the last *Send* event by the transitive closure of the relation $\texttt{R\_switch}_{tr}$, which relates *Send* events whose messages were delivered out of order at some location.

$$\begin{aligned}&i\ \texttt{R\_switch}_{tr}\ j \ \equiv\ \ \texttt{is-send}_{TE}(tr[i]) \ \wedge\ \texttt{is-send}_{TE}(tr[j])\\ &\qquad\wedge\ (i{<}j \wedge tr[j]{\downarrow}^{<}_{TE} tr[i] \ \vee\ j{<}i \wedge tr[i]{\downarrow}^{<}_{TE} tr[j])\end{aligned}$$

where $e{\downarrow}^{<}_{TE} tr[k]$ denotes that the event e is delivered at some time before time k. It is fairly easy to prove that Q has the first four of the five required properties. For the fifth, however, we need more than just $\texttt{switch\_inv}_{TE}$ and $\texttt{Causal}_{TE}$. We have to assume that the trace tr has a certain normal form, in which *Send* events occur as late as possible and asynchronous *Deliver* events match the order of their *Send* events. Since the algorithm for generating this normal form can be shown to preserve asynchronous and delayable properties, we call this property an *asynchronous-delayable normal form* or *AD-normal*.

$$\begin{aligned}&\texttt{AD-normal}_{TE}(tr)\\ &\equiv\ \forall \texttt{i}{<}|tr|.\ (\texttt{is-send}_{TE}(tr[\texttt{i}]) \wedge \neg\texttt{is-send}_{TE}(tr[\texttt{i+1}]) \ \Rightarrow\ tr[\texttt{i}] =^m_E tr[\texttt{i+1}]\\ &\qquad\wedge\ (\exists \texttt{j,k}{<}|tr|.\ \texttt{j}{<}\texttt{k} \ \wedge\ \texttt{is-send}_{TE}(tr[\texttt{j}]) \ \wedge\ tr[\texttt{j}]{\downarrow}_{TE} tr[\texttt{i+1}]\\ &\qquad\qquad\qquad\wedge\ \texttt{is-send}_{TE}(tr[\texttt{k}]) \ \wedge\ tr[\texttt{k}]{\downarrow}_{TE} tr[\texttt{i}] \ \ )\\ &\qquad\Rightarrow\ \ \texttt{loc}_{TE}(tr[\texttt{i}]) = \texttt{loc}_{TE}(tr[\texttt{i+1}]))\end{aligned}$$

This property, together with causality and no-duplicate-deliver suffices to prove the fifth property of Q. Thus altogether we have proved the following

$$\begin{aligned}&\vdash\ \forall \texttt{TE:TaggedEventStruct}\\ &\quad(\texttt{switch\_inv}_{TE} \ \wedge\ \texttt{Causal}_{TE} \ \wedge\ \texttt{AD-normal}_{TE} \ \wedge\ \texttt{No-replay}_{TE})\\ &\quad\triangleright\ \texttt{switch-decomposable}_{TE}\end{aligned}$$

4. Putting the above results together we can show

$$\vdash \forall \texttt{TE:TaggedEventStruct}.\ \forall \texttt{P:TraceProperty}_{TE}.\ \texttt{MCS}_{TE}\texttt{(P)} \ \wedge\ \texttt{P} \rhd (\texttt{Causal}_{TE} \wedge \texttt{No-replay}_{TE}) \Rightarrow (\texttt{switch\_inv}_{TE} \wedge \texttt{AD-normal}_{TE} \wedge \texttt{No-dup-send}_{TE})\ \texttt{fuses P}$$

In the last step we prove that the normal form requirement can be removed if we assume the predicate P to be asynchronous and delayable.
For this purpose we apply general theorem about the existence of partially sorted lists. It says that, if swapping adjacent elements e and e' in a list tr for which a decidable predicate $GOOD(e, e')$ does *not* hold increases the number of good pairs, then we may reach a list tr' in which all adjacent pairs satisfy $GOOD$ (proof by induction over the number of bad pairs).
By instantiating $GOOD$ with a localized version of $\texttt{AD-normal}_{TE}$ we can show that a trace can be converted into normal form.

$$\vdash \forall \texttt{TE:TaggedEventStruct}.\ \forall \texttt{tr:Trace}_{TE}\ \texttt{switch\_inv}_{TE}\texttt{(tr)} \wedge \texttt{No-dup-send}_{TE}\texttt{(tr)} \Rightarrow \exists \texttt{tr':Trace}_{TE}.\ \texttt{switch\_inv}_{TE}\texttt{(tr')} \wedge \texttt{tr}\ (\texttt{R\_delayable}_{TE} \vee \texttt{R\_asynch}_{TE})^*\ \texttt{tr'}$$

Since switchable properties are preserved by $\texttt{R\_delayable}_{TE} \vee \texttt{R\_asynch}_{TE}$, Theorem 1 follows from the above two results. □

The proof of Theorem 1 was developed completely within the NUPRL proof development system. Further details can be found in [5].

5.2 Global Correctness of Switching

Theorem 1 states that a switchable property P holds on a tagged event trace tr_m whenever P holds for all subtraces $tr_m|_{\texttt{tg}}$, provided tr_m satisfies the local switch invariant and does not contain duplicate *Send* events.

The *global switch invariant* expresses that some virtual inner trace tr_m, which is created by merging the traces tr_1 and tr_2 of the protocols below and is linked to the upper trace tr_u by introducing global and local delays and additional *Send* events, must satisfy the local switch invariant and be free of duplicates.

In the formal model, we describe the traces tr_1 and tr_2 by a single lower trace tr_l of tagged events. tr_l is related to tr_m by allowing adjacent events with different tags to be swapped, which accounts for the effects of buffering during *switch mode*. tr_m is related to tr_u by allowing (global and local) delays and enabling *Send* events.

$$\texttt{R}_{tag} \equiv (\texttt{swap-adjacent}_{[tag(e)\neq tag(e')]})^*$$
$$\texttt{R\_layer}_{TE} \equiv (\texttt{R\_async}_{TE} \vee \texttt{R\_delayable}_{TE} \vee \texttt{R\_send-enabled}_{TE})^*$$

where R^* denotes the transitive closure of a relation R. Note that $\texttt{R\_layer}_{TE}$ is the same as $\texttt{R\_layer}_{E}$ if $TE \in \texttt{TaggedEventStruct}_{E}$.

The definition of the global switch invariant summarizes all the above insights into a single relation between the lower and the upper trace.

$$\texttt{full\_switch\_inv}_{TE}(\texttt{tr}_u;\texttt{tr}_l) \equiv \exists \texttt{tr}_m\texttt{:Trace}_{TE}.\ \texttt{tr}_l\ \texttt{R}_{\texttt{tag}_{TE}}\ \texttt{tr}_m \wedge \texttt{switch\_inv}_{TE}(\texttt{tr}_m) \wedge \texttt{tr}_m\ \texttt{R\_layer}_{TE}\ \texttt{tr}_u \wedge \texttt{No-dup-send}_E(\texttt{tr}_u)$$

Switching protocols that implement this invariant, support all protocols that implement switchable properties. Whenever a switchable property P holds for all traces $tr_l|_{\texttt{tg}}$ of the individual protocols below a switching protocol that satisfies the global switching invariant, then it also holds for the trace tr_u above it. In the NUPRL proof development system this theorem, which is an straightforward consequence of Theorem 1, is formalized as follows.

Theorem 2 (Correctness of Switching).

$$\vdash\ \forall \texttt{E:EventStruct}.\ \forall \texttt{P:TraceProperty}_E.\ \forall \texttt{TE:TaggedEventStruct}_E\ \forall \texttt{tr}_u\texttt{:Trace}_E.\ \forall \texttt{tr}_l\texttt{:Trace}_{TE}.\ (\ \texttt{switchable}_E(\texttt{P}) \wedge \texttt{full\_switch\_inv}_{TE}(\texttt{tr}_u;\texttt{tr}_l)\ ) \Rightarrow (\forall \texttt{tg:Label}.\ \texttt{P}(\texttt{tr}_l|_{\texttt{tg}})) \Rightarrow \texttt{P}(\texttt{tr}_u)$$

5.3 Implementation Aspects

The implementation of a switching protocol was developed in parallel to the formal verification efforts, and took into account that the switch has to satisfy the global switch invariant. Together with the fact that switching protocols designed according to the description in Section 2 are *live*, this gives us assurance that the resulting hybrid protocol works correctly.

We have evaluated the performance implications of using the switching protocol by switching between two well-known total order protocols [19], one based on a centralized sequencer [16] and the other using a rotating token with a sequence number [8]. These two protocols have an interesting trade-off. The sequencer-based algorithm has low latency, but the sequencer may become a bottleneck when there are many active senders. The token-based algorithm does not have a bottleneck, but the latency is relatively high under low load. Experiments have shown that having the switch follow a small hysteresis at the cross-over point has the potential of achieving the best of both

6 Conclusion

We have designed a generic switching protocol for the construction of adaptive network systems and formally proved it correct with the NUPRL Logical Programming Environment. In the process we have developed an abstract characterization of communication properties that can be preserved by switching and an abstract characterization of invariants that an implementation of the switching protocol must satisfy in order to work correctly.

There has been a flurry of research on the verification of software systems and communication protocols (see e.g. [14, 21, 24, 26]). But most approaches so far provided only a-posteriori verifications of well-understood algorithms. To our knowledge this is the first case in which a new communication protocol was designed, verified, and implemented in parallel. Because of a team that consisted of both systems experts and experts in formal methods the protocol construction could proceed at the same pace of implementation as designs that are not formally assisted, and at the same time provide a formal guarantee for the correctness of the resulting protocol.

The verification efforts revealed a variety of implicit assumptions that are usually made when reasoning about communication systems and uncovered minor design errors that would have otherwise made their way into the implementation (such as to use switching for arbitrary protocols). This demonstrates that an expressive theorem proving environment with a rich specification language (such as provided by the NUPRL LPE) can contribute to the design and implementation of verifiably correct network software.

So far we have limited ourselves to investigating sufficient conditions for a switching protocol to work correctly. However, some of the conditions on switchable properties may be stricter than necessary. Reliability, for instance, is not a safety property, but we are confident that it is preserved by protocol layering and thus by our hybrid protocol. We intend to refine our characterization of switchable predicates and demonstrate that larger class of protocols can be supported.

Also, we would like to apply our proof methodology to the verification of protocol stacks. To prove that a given protocol stack satisfies certain properties, we have to be able to prove that these properties, once "created" by some protocol, are preserved by the other protocols in the stack. We believe that using meta-properties to characterize the properties preserved by specific communication protocols will make these investigations feasible.

Our case study has shown that formal methods are moving into the design and implementation phases of software construction as well as into the testing and debugging phases. The impact of formal methods is larger the more they are engaged at the earliest stages of design and implementation. We believe that the early use can add value to all subsequent stages, including the creation of informative documentation need for maintenance and evolution of software.

Acknowledgements This work was supported by DARPA grants F 30620-98-2-0198 (An Open Logical Programming Environment) and F 30602-99-1-0532 (Spinglass).

References

1. M. Aagaard & M. Leeser. Verifying a logic synthesis tool in Nuprl. In G. Bochmann & D. Probst, eds., *Workshop on Computer-Aided Verification*, LNCS 663, pages 72–83. Springer, 1993.
2. ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2`.
3. S. Allen, R. Constable, R. Eaton, C. Kreitz, L. Lorigo. The Nuprl open logical environment. In D. McAllester, ed., *17th Conference on Automated Deduction*, LNAI 1831, pages 170–176. Springer, 2000.

4. B. Alpern & F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
5. M. Bickford, C. Kreitz, R. van Renesse. Formally verifying hybrid protocols with the NUPRL logical programming environment. Technical report Cornell CS:2001-1839, Cornell University. Department of Computer Science, 2001.
6. K. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Co. & Prentice Hall, 1997.
7. K. Birman & R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.
8. J. Chang & N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, 1984.
9. E. M. Clarke, O. Grumberg, D. Peled. *Model Checking.* MIT Press, 1999.
10. R. Constable, S. Allen, M. Bromley, R. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, P. Mendler, P. Panangaden, J. Sasaki, S. Smith. *Implementing Mathematics with the* NUPRL *proof development system.* Prentice Hall, 1986.
11. M. Hayden. *The Ensemble System.* PhD thesis, Cornell University. Department of Computer Science, 1998.
12. J. Hickey, N. Lynch, R. van Renesse. Specifications and proofs for Ensemble layers. In R. Cleaveland, ed., *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 119–133. Springer, 1999.
13. D. Howe. Importing mathematics from HOL into NuPRL. In J. von Wright, J. Grundy, J. Harrison, eds., *Theorem Proving in Higher Order Logics*, LNCS 1125, pages 267–282. Springer, 1996.
14. D. Hutter, B. Langenstein, C. Sengler, J. H. Siekmann, W. Stephan, and A. Wolpers. Verification support environment (VSE). *Journal of High Integrity*, 1997.
15. Isabelle home page. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.
16. M. Kaashoek, A. Tanenbaum, S. Flynn-Hummel, H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, 1989.
17. C. Kreitz, M. Hayden, J. Hickey. A proof environment for the development of group communication systems. In C. & H. Kirchner, eds., *15^{th} Conference on Automated Deduction*, LNAI 1421, pages 317–332. Springer, 1998.
18. X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, R. Constable. Building reliable, high-performance communication systems from components. *Operating Systems Review* 34(5):80–92, 1999.
19. X. Liu, R. van Renesse, M. Bickford, C. Kreitz, R. Constable. Protocol switching: Exploiting meta-properties. In Luis Rodrigues & Michel Raynal, eds., *International Workshop on Applied Reliable Group Communication.* IEEE CS Press, 2001.
20. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
21. Z. Manna & A. Pnueli. *Temporal Verification of Reactive Systems.* Springer, 1995.
22. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.
23. Nuprl home page. `http://www.cs.cornell.edu/Info/Projects/NuPrl`.
24. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
25. PVS home page. `http://pvs.csl.sri.com`.
26. John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, 1999.
27. R. van Renesse, K. Birman, S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.
28. R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, D. Karr. Building adaptive systems using Ensemble. *Software—Practice and Experience*, 1998.