An Experiment in Formal Design Using Meta-properties

Mark Bickford, Christoph Kreitz, Robbert van Renesse, Robert Constable
Department of Computer Science,
Cornell-University,
Ithaca, NY 14853-7501
{markb,kreitz,rvr,rc}@cs.cornell.edu

Abstract

Formal methods tools have greatly influenced our ability to increase the reliability of software and hardware systems by revealing errors and clarifying critical concepts. In this article we show how a rich specification language and a theorem prover for it have contributed to the design and implementation of verifiably correct adaptive protocols. The protocol building team included experts in formal methods who were able to use the theorem prover to help guide protocol construction at the pace of implementation that is not formally assisted.

This example shows that formal methods can have a large impact when being engaged at the earliest stages of design and implementation, because they add value to all subsequent stages, including the creation of informative documentation needed for the maintenance and evolution of software.

1. Introduction

It is well established that formal methods contribute to our ability to build reliable software. Tools such as extended type checkers, model checkers and theorem provers 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.

In contrast to testing and a-posteriori verifications, which try to detect and eliminate errors in existing codes, formal methods used in the early stages of design and implementation cannot rely on details of the code, which is not yet written. Engagement of formal methods at this stage depends on the ability of the formal language to naturally and compactly express the ideas underlying the system.

At this early stage it is possible to state assumptions and goals that drive the system design. When it is possible to precisely define the concepts and goals, 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.

The formal process can proceed quickly if there is a sufficient database of basic facts about systems concepts already available and if these facts are ones that the design team uses in its discussions. We were able to achieve this state because of two years of prior work at the level of proving properties of protocols [13, 10, 3, 14]. This might be a natural evolution of formal methods, and if so it reveals additional value implicit in the work of falsification and verification, namely the task of building formal models and accumulating a body of formal facts.

In this paper we describe a case study in formal design that involves designing and implementing an adaptive network protocol for the Ensemble Group Communication system [9] using the NUPRL Logical Programming Environment (LPE) [8, 1] and the database of four thousand definitions, theorems and examples built up in the Common Logical Library of the LPE. Our design was centered around a characterization of communication properties that can be preserved by the adaptive protocol, which led to a study of *meta-properties* (i.e. properties of properties) as a means for classifying those properties.

We will first explore the formal background for designing communication systems and the representation of the corresponding concepts in the formal framework of the NUPRL LPE. We will then describe how our formal framework was used in the design of a verified hybrid protocol.

2. A formal model of communication

In order to support the formal design of protocols, we have developed a formal model of distributed communication systems and their properties. Our model formalizes notions for the specification of distributed algorithms in-

J. Lala, D. Maughan, C. McCollum, and B. Witten, eds, *DARPA Information Survivability Conference and Exposition (DISCEX-II)*, Anaheim, CA, June 2001. Volume II, pp. 100–107,

troduced by Lynch [16] and concepts used for the implementation of reliable network systems [6], particularly of Ensemble and its predecessors [5, 19, 9].

The model is based on the formal language of the NUPRL proof development system [8, 1], which already provides formalizations of the fundamental concepts of mathematics, data types, and programming. The NUPRL system supports conservative extensions of this language by user-defined concepts via abstractions and display forms. An *abstraction* of the form

$$\texttt{opid(parameters)} \equiv \texttt{expression with parameters}$$

defines a new (possibly parameterized) term of the formal language in terms of already existing formal 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 LPE supports interactive and tactic-based reasoning, 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. We have used this mechanism to create a technical report that provides a complete account of the formal work described in this paper [4].

2.1. 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 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.

For formal reasoning about events and traces we introduce a class `EventStruct` of formal *event structures*. An event structure $E \in \texttt{EventStruct}$ provides a carrier $\texttt{|}E\texttt{|}$ and three functions, $\texttt{is-send}_E$, $\texttt{loc}_E$, and $\texttt{msg}_E$, where

- $\texttt{is-send}_E(x)$ is `true` when the event $e \in \texttt{|}E\texttt{|}$ is a *Send* event (otherwise it is a *Deliver* event);
- $\texttt{loc}_E(e)$, the *location* of the event e, is the identifier of the process that sends or receives e; and
- $\texttt{msg}_E(e)$ is the message m 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. 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 simply a list of events of type $\texttt{|}E\texttt{|}$. The data type of traces over E is thus defined as

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

The requirement that traces should have no duplicate *Send* events will be formalized later as one of the properties of meaningful traces. All the usual list operations apply to traces as well, such as computing the length $\texttt{|}tr\texttt{|}$ of a trace tr, selecting the i-th element $tr[i]$ of tr, concatenation $tr_1@tr_2$ of two traces, the prefix relation $tr_1 \sqsubseteq tr_2$ between two traces, and filtering elements from a trace that satisfy a property P, denoted by $[e \in tr\,|\,P]$.

For process identifiers we introduce a (recursive) 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.

2.2. Properties of traces

A *trace property* is a predicate on traces that describes certain desired behaviors of communication. Typical examples are *Reliability* (every message that is sent is delivered to all receivers), *Integrity* (messages that are delivered have been sent by a trusted process), *Confidentiality* (non-trusted processes cannot see messages from trusted ones), or *Total Order* (processes that deliver the same two messages deliver them in the same order). We formalize trace properties as propositions on traces, i.e. as functions from $\texttt{Trace}_E$ to the type $\mathbb{P}$ of all logical propositions.

$$\texttt{TraceProperty}_E \equiv \texttt{Trace}_E \rightarrow \mathbb{P}$$

In this setting the properties *reliability* (for multicasting), *integrity*, *confidentiality*, and *total order* can be formalized as follows.

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

where `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 $tr_1 \big\downarrow_{tr_2}$ is the restriction of tr_1 to events whose messages also occur in tr_2.

$tr{\downarrow}p$
$\equiv$ `[ e` $\in tr$ `|` $\neg$`is-send`$_E$`(e)` $\wedge$ `loc`$_E$`(e)=`p `]`

$tr_1 \big\downarrow_{tr_2}$
$\equiv$ `[ e`$_1 \in tr_1$ `|` $\exists$`e`$_2 \in tr_2$`.` `e`$_1 =^m_E$ `e`$_2$ `]`

Three other properties are important as well. Every delivered message must have been sent before (*causality*), no message is sent twice, and no message is delivered twice (*replayed*) to the same process. These properties are assumed implicitly in the implementation of communication systems but need to be made explicit in a formal account.

`Causal`$_E(tr)$
$\equiv$ $\forall$`i<|`tr`|.`$\exists$`j`$\leq$`i.` tr`[j]` $=^m_E$ tr`[i]`
$\wedge$ `is-send`$_E(tr$`[j])`

`No-dup-send`$_E(tr)$
$\equiv$ $\forall$`i,j<|`tr`|.` `(` `is-send`$_E(tr$`[i])`
$\wedge$ `is-send`$_E(tr$`[j])`
$\wedge$ tr`[j]` $=^m_E$ tr`[i]`
`)` $\Rightarrow$ `i=j`

`No-replay`$_E(tr)$
$\equiv$ $\forall$`i,j<|`tr`|.` `(` $\neg$`is-send`$_E(tr$`[i])`
$\wedge$ $\neg$`is-send`$_E(tr$`[j])`
$\wedge$ tr`[j]` $=^m_E$ tr`[i]`
$\wedge$ `loc`$_E(tr$`[i])=loc`$_E(tr$`[j])`
`)` $\Rightarrow$ `i=j`

A *protocol* is a module available at every process that implements certain properties on behalf of the set of processes. It can be thought of as having a top and a bottom side, applications sitting at the top, and the network sitting at the bottom. Applications submit *Send* events to it, and the protocol submits *Send* events to the network below it. Vice versa, the network submits *Deliver* events to the protocol, and the protocol submits *Deliver* events to the application.

This symmetry makes it possible for protocols to be composed by layering them on top of one another. In effect, protocols are closed under composition: a *stack* of protocols is another protocol. It is also possible to view the application and the network as instances of protocols. In the context of a stack, we call a protocol a *layer*. Every process is required to have the same stack of layers.

2.3. 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 is a meta-property. But the meta-properties that we are interested in relate properties of the traces tr_u and tr_l above and below a protocol layer. We say that a reflexive and transitive relation R on traces *preserves* a property P if P holds for the trace tr_u, whenever the two traces tr_u and tr_l are related by R and P holds for tr_l. A similar definition is also given for ternary relations.

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

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

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

P `refines` Q
$\equiv$ $\forall$`tr:Trace`$_E$`.` P`(tr)` $\Rightarrow$ `Q(tr)`

Preservation by a relation R is a predicate on properties, i.e. a meta-property. Below, we will formalize four such relations that are important when dealing with protocols in any layered communication system. In section 4.1 we will discuss two additional relations that are necessary for the adaptive protocol.

Safety Safety [2] is probably the best-known meta-property. It means that a property does not depend on how far the communication has progressed: if the property holds for a trace, then it is also satisfied for every prefix of that trace. An example of a safe property is total order: taking events off the end of a trace cannot reorder message delivery. As an example of a property that is not safe, consider reliability. A reliable trace is one in which all sent messages have been delivered everywhere. However, if we chop off a suffix containing a *Deliver* event without the corresponding *Send* event, the resulting trace is no longer reliable.

The corresponding relation R for safety is R_{safety} which specifies that the upper trace is a prefix of the one below the protocol:

tr_u `R_safety`$_E$ tr_l
$\equiv$ $tr_u \sqsubseteq tr_l$

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.

The corresponding relation R_{asynch} specifies that two traces are related if they can be formed by swapping events that are adjacent and 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' that satisfy the condition $c(e; e')$.

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.

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 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.

The corresponding relation $R_{send-enabled}$ specifies that the upper trace is formed by adding *Send* events to the end of the lower trace.

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

3. The problem: building adaptive protocols

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. As a result, most existing adaptive protocols only adapt certain run-time parameters such as the flow window size in TCP [11] but not the overall behavior of the protocol, or focus on particular protocols such as flow control [17] or total order [18].

The approach of our systems Horus [20] and Ensemble [21, 9] is to *switch* between different protocols at run-time when necessary. However, it was never quite clear under what circumstances such a switch would actually preserve the properties of the individual protocols, i.e. how to guarantee that the result was actually *correct*.

The purpose of our experiment was 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 is supposed to interact with the application in a transparent fashion, that is, the application cannot tell easily that it is running on the switching protocol rather than on one of the underlying protocols, even as the switching protocol switches between protocols. The kinds of uses we envision include the following:

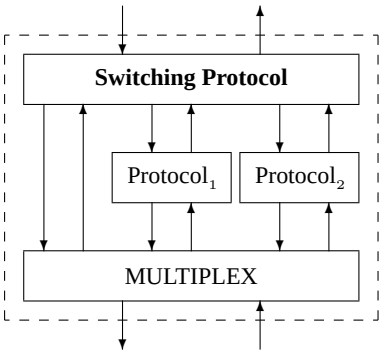
- *Performance.* By using the best protocol for a particular network and application behavior, performance can always be optimal.
- *On-line Upgrading.* Protocol switching can be used to upgrade networking protocols at run-time without having to restart applications. Even minor bug fixes may be done in this way.
- *Security.* System managers will be able to increase security at run-time, for example when an intrusion detection system notices unusual behavior, or when it gets close to April 1st.

To guarantee that the switching protocol preserves a variety of communication properties, formal methods were used early in the design phase. In addition to the four meta-properties discussed so far we have developed two meta-properties to classify properties that are *switchable* at all. To design the switching protocol correctly, we have characterized the *invariants* it has to satisfy and proved that they are sufficient to preserve switchable properties.

4. Formal design of hybrid protocols

The basic idea of the switching protocol is to operate in one of two modes. In *normal mode* the switching protocol simply forwards messages from the application to the current protocol and vice versa. Should there be a need to switch to a different protocol, the switching protocol goes into *switching mode*, during which any process will deliver all messages for the previous protocol while buffering messages that are to be delivered for the new one. The switching protocol will return to normal mode as soon as all messages for the previous protocol have been delivered.

The switching protocol will reside on top of the individual protocols, coupled by a multiplexer below them, as illustrated in the following diagram.



To prove that the resulting hybrid protocol preserves the specification of the individual protocols we proceed in two phases. We first give an abstract classification of *switchable* communication properties and develop a *switching invariant* that a protocol with the above architecture must satisfy in order to preserve switchable properties. We then develop and formalize a concrete switching protocol that preserves the switching invariant.

4.1. Meta-properties of hybrid protocols

The four meta-properties discussed so far, *Safety*, *Asynchrony*, *Send Enabled*, and *Delayable*, are sufficient for properties to survive the effects of delay in a layered environment. Since the switching protocol does introduce delays, these meta-properties are going to be important for a property to be preserved by the switching protocol.

1. Liveness properties require that the input satisfy some fairness condition. Since the switching protocol divides the input between the two protocols, *safety* can guarantee that the fairness condition holds.
2. *Asynchrony* is needed because delays in distributed systems can re-order global orderings.
3. *Delayable* is needed because the switching protocol will introduce a delay that can re-order local orderings.
4. *Send Enabled* is needed because any restriction on the relative order of sending is lost when we switch between protocols.

In addition to these meta-properties we need two meta-properties, which express that properties shall be preserved under switching. These will be discussed below.

Memoryless When we switch between protocols, the current protocol may not see part of the history of events that were handled by a different protocol. It thus has to be able to work *memoryless*, i.e. as if these events never happened.

A property is *memoryless* if we can remove all events pertaining to a particular message from 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* (i.e. without *local* memory) 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.

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.

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

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 their concatenation. Total order is composable, because the concatenation of traces does not change the order of events in either trace.

The corresponding relation $R_{composable}$ is ternary, as it characterizes the upper trace tr as concatenation of two lower traces without common messages.

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

4.2. 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 [1] required us to make many assumptions explicit that are usually implicitly present in an informal analysis of communication protocols. In the process we have refined the notion of switchability until it was formally strong enough for a verification of the switching protocol while being expressed in terms of concepts that are natural to communication systems.

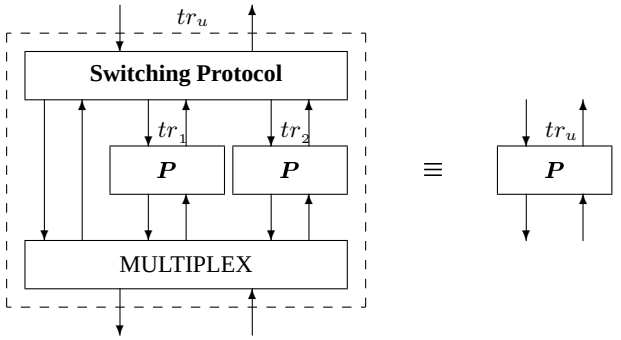
A trace property P is *switchable* if it satisfies all of the six meta-properties and requires the trace to be meaningful, i.e. that delivered messages were actually sent and never delivered twice to the same process.

$$\begin{aligned}
&\mathtt{switchable}_E(P) \\
\equiv\ & P \ \mathtt{refines}\ \mathtt{Causal}_E \\
\wedge\ & P \ \mathtt{refines}\ \mathtt{No\text{-}replay}_E \\
\wedge\ & \mathtt{R_safety}_E \ \mathtt{preserves}\ P \\
\wedge\ & \mathtt{R_async}_E \ \mathtt{preserves}\ P \\
\wedge\ & \mathtt{R_delayable}_E \ \mathtt{preserves}\ P \\
\wedge\ & \mathtt{R_send\text{-}enabled}_E \ \mathtt{preserves}\ P \\
\wedge\ & \mathtt{R_memoryless}_E \ \mathtt{preserves}\ P \\
\wedge\ & \mathtt{R_composable}_E \ \mathtt{preserves}_3\ P
\end{aligned}$$

4.3. The switching invariant

While switchability is an abstract characterization of communication protocols whose properties can be preserved by switching, the switching invariant is an abstract characterization of the switching protocol that an implementation must satisfy if it shall be guaranteed to preserve switchable properties. Like the switchability meta-property, it is the result of several refinements of a design process that was guided by verification.

In order to prove the switching protocol preserves a property P we have to show that P holds for the trace tr_u above the switch whenever it holds for the traces tr_1 and tr_2 of the two protocols below. 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 switching protocol affects the traces tr_1 and tr_2 in two ways: first, they 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 have introduced a *virtual middle trace* tr_m that consists of the events of tr_1 and tr_2. We have developed a *local switch invariant*, which tr_m must satisfy to guarantee that a property holds on tr_m whenever it holds on its subtraces. From that we have derived a (global) *full switch invariant* by linking tr_m to tr_1 and tr_2 via merging and by linking tr_m to tr_u through the introduction of global and local delays and additional *Send* events. The full switch invariant models the basic architecture of the switching protocol described at the beginning of this section and has formally been proven to guarantee its correctness.

In order to identify the origin of events in a merged trace we define a class `TaggedEventStruct` of *tagged event structures*. A tagged event structure $TE \in$ `TaggedEventStruct` provides the same components as any element of `EventStruct` but an additional function $\mathtt{tag}_{TE}$ that computes the label $tg \in$ `Label` of an event $e \in |TE|$. By $\mathtt{TaggedEventStruct}_E$ 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

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

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

The local switch invariant shall guarantee that a switchable property P holds for tr_m whenever P holds for all subtraces $tr_m|_{tg}$. From the description of the switching protocol we know that if two messages are sent using different protocols, then each process buffers the second message 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.

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

where $e\downarrow_{TE} tr\mathtt{[}k\mathtt{]}$ denotes that an event e is delivered at time k in tr:

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

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

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 while leaving the order of events with a given tag unchanged, 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. Furthermore, the upper trace must be free of duplicate *Send* events.

$$\begin{aligned}
&\mathtt{full_switch_inv}_{TE}(\mathtt{tr}_u;\mathtt{tr}_l)\\
&\equiv\ \exists \mathtt{tr}_m\mathtt{:Trace}_{TE}.\ \ \mathtt{tr}_l\ \mathtt{R}_{\mathtt{tag}_{TE}}\ \mathtt{tr}_m\\
&\qquad\qquad \wedge\ \mathtt{switch_inv}_{TE}(\mathtt{tr}_m)\\
&\qquad\qquad \wedge\ \mathtt{tr}_m\ \mathtt{R_layer}_{TE}\ \mathtt{tr}_u\\
&\qquad\qquad \wedge\ \mathtt{No\text{-}dup\text{-}send}_E(\mathtt{tr}_u)
\end{aligned}$$

where the relations $\mathtt{R}_{tag}$ and $\mathtt{R\_layer}_{TE}$ are defined as follows (R^* denotes the transitive closure of a relation R).

$$\mathtt{R}_{tag} \equiv (\mathtt{swap\text{-}adjacent}_{[tag(e)\neq tag(e')]})^*$$

$$\mathtt{R\_layer}_{TE} \equiv (\ \mathtt{R\_async}_{TE} \vee \mathtt{R\_delayable}_{TE} \vee \mathtt{R\_send\text{-}enabled}_{TE}\ )^*$$

4.4. Proving hybrid protocols correct

Using the NUPRL theorem prover [1] we have shown that switching protocols that satisfy the full switching invariant can support those protocols that implement switchable properties. Whenever a trace property P is switchable and holds for all traces $tr_l|_{\mathtt{tg}}$ of the individual protocols below the switching protocol, then it also holds for the trace tr_u above the switching protocol, provided that the switching protocol satisfies the global switching invariant. In the NUPRL system this theorem is formalized as follows.

Theorem (Correctness of Switching)

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

The proof of this theorem proceeds by induction is based on a complex series of intermediate lemmata that refine the prerequisites for preserving certain classes of properties. These lemmata eventually lead to a proof that the local switch invariant suffices to preserve switchable predicates on the virtual middle trace: a switchable predicate P holds for tr_m whenever it holds for all traces $tr_l|_{\mathtt{tg}}$ and tr_m satisfies the local switch invariant. We then prove that P is preserved by the tag-relation between tr_l and tr_m and layer relation between tr_m and tr_u.

The proof, whose details can be found in [4], was developed completely within the NUPRL LPE and thus provides a formal verification of the switching protocol.

4.5. Implementing the switching protocol

The switching invariant characterizes properties that a switching protocol has to implement in order to preserve switchable properties of protocols below it. We now describe a particular switching protocol that has been designed in the process of this experiment and satisfies the switching invariant.

As mentioned above, this switching protocol has two modes of operation. In *normal mode*, when the application submits a message for sending to the switching protocol, the switching protocol in turn offers the message to the current protocol. Whenever receiving a message from the current protocol, the switching protocol simply forwards the message to the application. However, when there is a request to switch, the switching protocol goes into *switching mode*.

First, one of the processes called the *manager* broadcasts a *PREPARE* message to the other members. On receipt, a member returns an *OK* message that includes the number of messages that the member has sent so far over the current protocol. New data messages will be sent over the new protocol, but messages received over this protocol will be buffered.

The manager awaits all *OK* messages, and then broadcasts a *SWITCH* message, including a vector with the message-send count of each member. On receipt, a member knows how many messages it should have delivered from each other member. When it has received and delivered all messages of the current protocol from each member, the member switches over to the new protocol and delivers any messages that were buffered.

To avoid congestion on the network, our implementation of the switching protocol does not actually do network-level broadcasts, but rotates a token message in a logical ring of the group members. We have evaluated the performance implications of using our switching protocol by switching between two well-known mechanisms for implementing total order, one based on a centralized sequencer [12] and the other using a rotating token with a sequence number [7].

These two mechanisms 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 since processes have to await the token before they can send. A hybrid protocol formed by switching at the cross-over point has the potential of achieving the best of both worlds. However, some care needs to be taken in practice, as the overhead of switching depends on the latency of the current protocol. Experiments have shown that adding a small hysteresis leads to the best practical results [15].

5. 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.

Our characterization gives 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 as well.

Our 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. This demonstrates that formal reasoning about group communication in an expressive theorem proving environment such as the NUPRL Logical Programming Environment can contribute to the design and implementation of hybrid protocols.

Because our team consisted of both systems experts and experts in formal methods the protocol construction and implementation could proceed at the same pace as designs that are not formally assisted while providing a formal guarantee for the correctness of the resulting protocol.

Our experiment shows 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 needed for maintenance and evolution of software.

Acknowledgements

Part of 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] S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The NUPRL open logical environment. In D. McAllester, editor, *17^{th} Conference on Automated Deduction*, *Lecture Notes in Artificial Intelligence* 1831, pages 170–176. Springer, 2000.

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[3] M. Bickford and J. Hickey. Predicate transformers for infinite-state automata in NUPRL type theory. In *Irish Formal Methods Workshop*, 1999.

[4] M. Bickford, C. Kreitz, and R. van Renesse. Formally verifying hybrid protocols with the NUPRL logical programming environment. Technical report, Cornell University. Department of Computer Science, 2001.

[5] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[6] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Co. & Prentice Hall, 1997.

[7] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, 1984.

[8] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the* NUPRL *proof development system*. Prentice Hall, 1986.

[9] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University. Department of Computer Science, 1998.

[10] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In R. Cleaveland, editor, *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 1579, pages 119–133. Springer, 1999.

[11] V. Jacobson. Congestion avoidance and control. In *Symposium on Communications Architectures & Protocols*, Stanford, CA, 1988. ACM SIGCOMM.

[12] M. F. Kaashoek, A. S. Tanenbaum, S. Flynn-Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, 1989.

[13] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *15^{th} Conference on Automated Deduction*, *Lecture Notes in Artificial Intelligence* 1421, pages 317–332. Springer, 1998.

[14] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *17^{th} ACM Symposium on Operating Systems Principles (SOSP'99)*, *Operating Systems Review*, 34(5):80–92, 1999.

[15] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. In L. Rodrigues and M. Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*. IEEE Computer Society Press, 2001.

[16] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[17] P. K. McKinley, R. T. Rao, and R. F. Wright. H-RMC: A hybrid reliable multicast protocol for the Linux kernel. In *Conference on Supercomputing '99*, 1999.

[18] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *16th International Conference on Distributed Computing Systems*. IEEE CS Press, 1996.

[19] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[20] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A Framework for Protocol Composition in Horus. In *14^{th} ACM Symposium on Principles of Distributed Computing*, pages 80–89, 1995. ACM SIGOPS-SIGACT.

[21] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr. Building adaptive systems using Ensemble. *Software—Practice and Experience*, 1998.