Formally Verifying Hybrid Protocols with the Nuprl Logical Programming Environment

Mark Bickford, Christoph Kreitz, Robbert van Renesse

We describe a generic switching protocol for the construction of hybrid protocols and prove it correct with the NUPRL proof development system. We introduce the concept of *meta-properties* to characterize communication properties that can be preserved by switching and identify *switching invariants* that an implementation of the switching protocol must satisfy in order to work correctly.
Our work shows how a theorem prover with a rich specification language can contribute to the design and implementation of verifiably correct adaptive protocols and that it can have a large impact when being engaged at the earliest stages of the design.

Technical Report

Department of Computer Science
Cornell University
Ithaca, New York

February, 2001

Contents

Formally Verifying Hybrid Protocols with the Nuprl Logical Programming Environment*

Mark Bickford, Christoph Kreitz, Robbert van Renesse

Department of Computer Science, U.S.A.
Cornell University,
Ithaca, NY, 14853

{markb,kreitz,rvr}@cs.cornell.edu

February 24, 2001

Abstract

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

Our work shows how a theorem prover with a rich specification language can contribute to the design and implementation of verifiably correct adaptive protocols and that it can have a large impact when being engaged at the earliest stages of the design.

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 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.
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.

*This technical report was created automatically from formal NUPRL objects using NUPRL's formal documentation mechanism. It has not been edited except for minor formatting adjustments to correct overwide lines.

The NUPRL Logical Programming Environment (LPE) [4, 2] 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 [7] as well as the verification and optimization of communication protocols [9, 6, 3, 8, 10].
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 [12, 5, 11]. 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 preserved by switching. This led to a study of *meta-properties*, i.e.$\hat{=}$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. properties of communication protocols that can be preserved by switching (Section 4.5). In Section 4 we will 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 5.3 we develop the switch-invariant for switching protocols and formally prove that switchable properties are preserved by switching whenever the 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.
The Ensemble system [12, 5] 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 stacking architecture like the one used in Ensemble the switching protocol will reside on top of the individual protocols.

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 protocol, the *SP* goes into {\ef 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 *SP* will return to normal mode as soon as all messages for the previous protocol have been delivered.

The above description served as the starting point for proving the correctness of the resulting hybrid protocol and subsequently for the implementation of the switching protocol as well. Our verification proceeds in two phases. We first classify communication properties that are *switchable*, i.e. have the potential to be preserved under switching, and then derive a *switching invariant* that a switching protocol must satisfy to preserve switchable properties.

3 Notation and Prerequisites

This paper was created as an object in the Nuprl library. All the lemmas and theorems quoted in this paper, in fact, everything that appears in `typewriter font` is the display form of a Nuprl term. The careful reader will notice that some of our definitions use parameters that are not shown in their display form. In such cases, we have chosen a display form for the term that suppresses a parameter because we think the parameter will be clear from context. In a session with the Nuprl system, a user can always use an alternate display form that shows all the parameters, but for this printed document we have chosen the forms that seem most readable without loss of information. We have actually left more parameters exposed than necessary; many of the definitions that we will present have a parameter `E` that is an *event structure* and is displayed as a subscript. In all of our theorems, there is only one relevant event structure `E`, so we could have supressed the parameter `E` in the display of terms that mention it.

Our logical notation is standard, so only one comment on logical notation is called for. When stating the theorem that propositions `A`,`B`, and `C` imply proposition `D`, we write

```
A ⇒ B ⇒ C ⇒ D
```

rather than

```
(A ∧ B ∧ C) ⇒ D
```

We do this because, in constructive logic, every theorem has an extract and the extract of the first form is a Curried function while the extract of the second form is uncurried. Our tactics work best on the Curried form. In the proofs given in this paper (which are English summaries of the Nuprl proofs), we will refer to `D` as "the conclusion" and to `B` as "the second hypothesis".

The type theory prerequisites are minimal. We will mention only that Nuprl type theory includes intersection types and void types, and this lets us define the type `Top` by:

```
Top == ∩x:Void.Void
```

The only thing we need to know about this type is that every type `T` is a subtype of `Top`. In our treatment of structures, we use the dependent product type

$$x{:}T_1 \times T_2(x)$$

We will also remind readers that in constructive type theory, propositions $\mathbb{P}$ are types, not booleans $\mathbb{B}$, and that we have $P \vee (\neg P)$ only for *decidable* propositions. For this paper, we can mostly ignore this distinction because whenever we need to know that a proposition is decidable, Nuprl's Auto-tactic can prove it for us.
We will use a number of operations on lists and will use facts about them without proof. The real Nuprl proofs come from an extensive list-theory library. Here, we will merely show the notations we use for list operations. In a list type `T List`, we have nil and cons constructors, `[]` and `[a / L]`, and lists built by cons-ing to nil will display like `[x; y; z]`. The length and append functions are $\|L\|$ and L_1 @ L_2. The type $\mathbb{N}_{\|L\|}$ of natural numbers less than the length of `L` is the domain of the selection function `L[i]`. The prefix, or initial-segment, relation is written $L_1 \leq L_2$. The filter operation forms lists like $<x \in L \mid P(x)>$, the list obtained by swapping the elements of `L` at indices `i` and `j` is `swap(L;i;j)` and we define the *swap adjacent* relation on lists by:

$$L1 \text{ swap-adjacent}_{P(x;\ y)} \ L2 ==$$
$$\exists i{:}\mathbb{N}_{\|L1\|-1}.\ P(L1[i];\ L1[i+1]) \wedge (L2 = \text{swap}(L1;i;i+1))$$

Notice that, as in the lefthand side of this definition, we like to use infix notation for the application of a relation.

4 Formal Model of Traces, Properties, & Meta-properties

4.1 Structures

The formal model of many concepts consists of a type, operations defined over that type, and assumptions (axioms) about the operations. We like to package the type and its operations and assumptions into one formal object that we call a *structure*. Every structure `M` is a tuple whose first component, which we write as `|M|` and call the *carrier* of `M`, is a type, and whose other components are functions defined over `|M|` or propositions about these functions. Thus every structure is a member of the type `Structure` defined as

$$\text{Structure} == T{:}\mathbb{U} \times \text{Top}$$

The second component type `Top` allows us to form subtypes of `Structure` by replacing `Top` with other types. For example, we define the structure of a decidable equivalence relation as follows

$$\text{DecidableEquiv} == T{:}\mathbb{U} \times E{:}T \to T \to \mathbb{B} \times \text{EquivRel}(T)((_1\ E\ _2)) \times \text{Top}$$

If `D` is a member of `DecidableEquiv` the the second component of `D`, which we write as $=_D$, is a binary boolean relation on `|D|`. The third component is a witness that $=_D$ is an equivalence relation on `|D|` and the final `Top` in the structure allows us to form subtypes of `DecidableEquiv` that have additional operations or assumptions.

4.2 Messages

Processes multicast messages. Each message will have a content and a sender. Messages will also have a unique id, so that messages with the same content and sender can be distinguished. To model messages we introduce the type of message structures

$$\texttt{MessageStruct} == \texttt{M}{:}\mathbb{U} \times \texttt{C:DecidableEquiv} \times \texttt{M} \to \texttt{|C|} \times \texttt{M} \to \texttt{Label} \times \texttt{M} \to \mathbb{Z} \times \texttt{Top}$$

If $\texttt{M} \in \texttt{MessageStruct}$, then its carrier $\texttt{|M|}$ is the type of messages. The third component, which we write $\texttt{content}_{\mathrm{M}}$ is a map from the messages $\texttt{|M|}$to their content $\texttt{|C|}$, which is the carrier of a decidable equivalence relation, $\texttt{cEQ}_{\mathrm{M}}$, given by the second component of the message structure. This gives us a way to decide when the contents of two messages are "the same". The forth and fifth components, $\texttt{sender}_{\mathrm{M}}$ and $\texttt{uid}_{\mathrm{M}}$ are the operations that map messages to their sender and unique id. The two messages are equal if they have the same content, sender, and id, so we define

$$\begin{aligned} \texttt{m}_1 =_{\mathrm{M}} \texttt{m}_2 == \; & (\texttt{content}_{\mathrm{M}}(\texttt{m}_1) =_{\texttt{cEQ}_{\mathrm{M}}} \texttt{content}_{\mathrm{M}}(\texttt{m}_2)) \\ \wedge \; & (\texttt{sender}_{\mathrm{M}}(\texttt{m}_1) = \texttt{sender}_{\mathrm{M}}(\texttt{m}_2)) \\ \wedge \; & (\texttt{uid}_{\mathrm{M}}(\texttt{m}_1) = \texttt{uid}_{\mathrm{M}}(\texttt{m}_2)) \end{aligned}$$

4.3 Events and Traces

We say that $\texttt{E}$ is an $\texttt{EventStruct}$ if it provides a type $\texttt{|E|}$ of events, a message structure, $\texttt{MS}_{\mathrm{E}}$, and three functions, $\texttt{msg}_{\mathrm{E}}$, $\texttt{loc}_{\mathrm{E}}$, and $\texttt{is-send}_{\mathrm{E}}$. When $\texttt{is-send}_{\mathrm{E}}\texttt{(e)}$ is true we say that event $\texttt{e} \in \texttt{|E|}$ is a send event; otherwise we call it a deliver event and write $\texttt{is-deliver}_{\mathrm{E}}\texttt{(e)}$. The location of event $\texttt{e}$ is $\texttt{loc}_{\mathrm{E}}\texttt{(e)}$, and its message content is $\texttt{msg}_{\mathrm{E}}\texttt{(e)}$ which is a member of $\texttt{|MS}_{\mathrm{E}}\texttt{|}$. Using these functions we define the binary relation, $\texttt{e}_1 \texttt{ =msg=}_{\mathrm{E}} \texttt{ e}_2$, that holds when events $\texttt{e}_1$ and $\texttt{e}_2$ have the same message content. For example, $\texttt{e}_1$ and $\texttt{e}_2$ might be delivery events of a message $\texttt{m}$ at two different locations. We can show that this relation is an equivalence relation on events.

$$\texttt{e}_1 \texttt{ =msg=}_{\mathrm{E}} \texttt{ e}_2 == \texttt{msg}_{\mathrm{E}}(\texttt{e}_1) =_{\texttt{MS}_{\mathrm{E}}} \texttt{msg}_{\mathrm{E}}(\texttt{e}_2)$$

The formal definition of the type of event structures is

$$\texttt{EventStruct} == \texttt{E}{:}\mathbb{U} \times \texttt{M:MessageStruct} \times \texttt{E}{\to}\texttt{|M|} \times \texttt{E}{\to}\texttt{Label} \times \texttt{E}{\to}\mathbb{B} \times \texttt{Top}$$

Given an event structure, E, a *trace* is just a list of events

$$\texttt{Trace}_{\mathrm{E}} == \texttt{|E| List}$$

A trace $\texttt{tr}$ defines an ordering of the events in it. We also call the list indices of $\texttt{tr}$, the members of $\mathbb{N}_{||\texttt{tr}||}$, *times* and we say that event $\texttt{tr[k]}$ occured at time $\texttt{k}$. The message in event $\texttt{x}$ was delivered at time $\texttt{k}$ if

$$\texttt{x delivered at time k} == (\texttt{x =msg=}_{\mathrm{E}} \texttt{ tr[k]}) \wedge \texttt{is-deliver}_{\mathrm{E}}\texttt{(tr[k])}$$

If the message in event x was delivered at some location earlier than any delivery of the message in event y at that same location, then

$$\begin{aligned}&\texttt{x somewhere delivered before y ==}\\&\quad \exists \texttt{k}:\mathbb{N}_{\|\texttt{tr}\|}.\ \ \texttt{x delivered at time k}\\&\qquad\qquad \wedge\ (\forall \texttt{k}':\mathbb{N}_{\|\texttt{tr}\|}.\ \texttt{y delivered at time k}'\\&\qquad\qquad\qquad\qquad \Rightarrow\ (\texttt{loc}_{\mathrm{E}}(\texttt{tr[k}'\texttt{]}) = \texttt{loc}_{\mathrm{E}}(\texttt{tr[k]}))\ \Rightarrow\ (\texttt{k} \le \texttt{k}'))\end{aligned}$$

Here is a simple lemma about this relation that we will need later.

Lemma 4.1

$$\begin{aligned}&\forall \texttt{E:EventStruct}.\ \forall \texttt{a,b,c}:|\texttt{E}|.\ \forall \texttt{tr}:|\texttt{E}|\ \texttt{List}.\\&\quad \texttt{a somewhere delivered before b}\\&\quad \Rightarrow\ (\texttt{a somewhere delivered before c}\ \vee\ \texttt{c somewhere delivered before b})\end{aligned}$$

Proof: If a somewhere delivered before b then there is a time k and a location, p = $\texttt{loc}_{\mathrm{E}}$(tr[k]) such that the message in event a was delivered to location p at time k but no delivery of the message in event b to location p has occured before time k. If a delivery of the message in event c to location p has occured before time k, then c somewhere delivered before b. If not, then a somewhere delivered before c. This case split is decidable, so the conclusion follows □

4.4 Trace Properties and Refinement

A trace property is a proposition on traces

$$\texttt{TraceProperty}_{\mathrm{E}}\ \texttt{==}\ |\texttt{E}|\ \texttt{List} \rightarrow \mathbb{P}$$

The following is the definition of the refinement relation on trace properties. It is read "property P *refines* property Q".

$$\texttt{P} \rhd \texttt{Q}\ \texttt{==}\ \forall \texttt{tr}:|\texttt{E}|\ \texttt{List}.\ \texttt{P(tr)}\ \Rightarrow\ \texttt{Q(tr)}$$

Every property refines property $\texttt{P}_{\texttt{True}}$ defined by

$$\texttt{P}_{\texttt{True}}\texttt{(tr) == True}$$

Here are three trace properties that we will need in the sequel.

$$\begin{aligned}&\texttt{Causal}_{\mathrm{E}}\texttt{(tr) ==}\\&\quad \forall \texttt{i}:\mathbb{N}_{\|\texttt{tr}\|}.\ \exists \texttt{j}:\mathbb{N}_{\|\texttt{tr}\|}.\ ((\texttt{j} \le \texttt{i})\ \wedge\ \texttt{is-send}_{\mathrm{E}}(\texttt{tr[j]}))\ \wedge\ (\texttt{tr[j] =msg=}_{\mathrm{E}}\ \texttt{tr[i]})\\&\texttt{No-dup-send}_{\mathrm{E}}\texttt{(tr) ==}\\&\quad \forall \texttt{i,j}:\mathbb{N}_{\|\texttt{tr}\|}.\ \ \texttt{is-send}_{\mathrm{E}}(\texttt{tr[i]})\\&\qquad\qquad \Rightarrow\ \texttt{is-send}_{\mathrm{E}}(\texttt{tr[j]})\\&\qquad\qquad \Rightarrow\ (\texttt{tr[i] =msg=}_{\mathrm{E}}\ \texttt{tr[j]})\\&\qquad\qquad \Rightarrow\ (\texttt{i = j})\\&\texttt{No-dup-deliver}_{\mathrm{E}}\texttt{(tr) ==}\\&\quad \forall \texttt{i,j}:\mathbb{N}_{\|\texttt{tr}\|}.\ \ (\neg\texttt{is-send}_{\mathrm{E}}(\texttt{tr[i]}))\\&\qquad\qquad \Rightarrow\ (\neg\texttt{is-send}_{\mathrm{E}}(\texttt{tr[j]}))\\&\qquad\qquad \Rightarrow\ (\texttt{tr[j] =msg=}_{\mathrm{E}}\ \texttt{tr[i]})\\&\qquad\qquad \Rightarrow\ (\texttt{loc}_{\mathrm{E}}(\texttt{tr[i]}) = \texttt{loc}_{\mathrm{E}}(\texttt{tr[j]}))\\&\qquad\qquad \Rightarrow\ (\texttt{i = j})\end{aligned}$$

4.5 Meta-properties

We classify trace properties using meta-properties. The meta-properties we need are all instances of the following schemas.

$$
\begin{array}{lll}
\mathtt{R\ preserves\ P} & \mathtt{==} & \forall \mathtt{x,y:Trace}_{\mathrm{E}}.\ \mathtt{P(x)} \Rightarrow \mathtt{(x\ R\ y)} \Rightarrow \mathtt{P(y)} \\
\mathtt{(ternary)\ R\ preserves\ P} & \mathtt{==} & \forall \mathtt{x,y,z:Trace}_{\mathrm{E}}.\ \mathtt{P(x)} \Rightarrow \mathtt{P(y)} \Rightarrow \mathtt{R(x,y,z)} \Rightarrow \mathtt{P(z)} \\
\mathtt{P} \rhd \mathtt{Q} & \mathtt{==} & \forall \mathtt{tr:|E|\ List.\ P(tr)} \Rightarrow \mathtt{Q(tr)}
\end{array}
$$

A trace property is a *safety* property if it is preserved by

$$\mathtt{tr}_1\ \mathtt{safetyR}_{\mathrm{E}}\ \mathtt{tr}_2\ \mathtt{==}\ \mathtt{tr}_2 \leq \mathtt{tr}_1$$

A property is *memoryless* if it is preserved by the operation of removing all events with a given message. So memoryless properties are the ones that are preserved by

$$\mathtt{L}_1\ \mathtt{memorylessR}_{\mathrm{E}}\ \mathtt{L}_2\ \mathtt{==}\ \exists \mathtt{a:|E|.\ L}_2 = \mathtt{<b} \in \mathtt{L}_1\ \mathtt{|}\ \neg\mathtt{(b\ =msg=}_{\mathrm{E}}\ \mathtt{a)>}$$

A property is *send-enabled* if it is preserved when a send event is appended to the trace. Send-enabled properties are the ones preserved by

$$\mathtt{L}_1\ \mathtt{send\text{-}enabledR}_{\mathrm{E}}\ \mathtt{L}_2\ \mathtt{==}\ \exists \mathtt{x:|E|.\ is\text{-}send}_{\mathrm{E}}\mathtt{(x)} \wedge \mathtt{(L}_2 = \mathtt{(L}_1\ \mathtt{@\ [x]))}$$

A property is *asynchronous* if it is preserved when adjacent deliver events or adjacent send events that have different locations are swapped. Asynchronous properties are the ones preserved by

$$\mathtt{asyncR}_{\mathrm{E}}\ \mathtt{==}\ \mathtt{swap\text{-}adjacent}_{\mathtt{((\neg(loc_{E}(x)\ =\ loc_{E}(y)))}}$$

A property is *delayable* if it is preserved when adjacent events, one of which is a send and the other a deliver, and which have different message content, are swapped.

$$
\begin{array}{l}
\mathtt{x\ R\_del}_{\mathrm{E}}\ \mathtt{y}\ \mathtt{==} \\
\qquad \mathtt{(\neg(x\ =msg=}_{\mathrm{E}}\ \mathtt{y))} \\
\quad \wedge\ \mathtt{((is\text{-}deliver}_{\mathrm{E}}\mathtt{(x)} \wedge \mathtt{is\text{-}send}_{\mathrm{E}}\mathtt{(y))} \vee \mathtt{(is\text{-}send}_{\mathrm{E}}\mathtt{(x)} \wedge \mathtt{is\text{-}deliver}_{\mathrm{E}}\mathtt{(y)))} \\
\mathtt{delayableR}_{\mathrm{E}}\ \mathtt{==}\ \mathtt{swap\text{-}adjacent}_{\mathtt{x\ R\_del_{E}\ y}}
\end{array}
$$

A property is *composable* if it is preserved when two traces, L_1 and L_2, that have no messages in common, are appended. Composable properties are the ones preserved by the ternary relation

$$\mathtt{composableR}_{\mathrm{E}}\mathtt{(L}_1\mathtt{,L}_2\mathtt{,L)}\ \mathtt{==}\ \mathtt{(}\forall \mathtt{x} \in \mathtt{L}_1.\ \forall \mathtt{y} \in \mathtt{L}_2.\ \neg\mathtt{(x\ =msg=}_{\mathrm{E}}\ \mathtt{y))} \wedge \mathtt{(L = (L}_1\ \mathtt{@\ L}_2\mathtt{))}$$

4.6 Tagged Events

When we reason about a combination of protocols, we associate a label with each protocol and tag the events handled by each protocol with that protocol's label. To model these tagged events, we make a subtype of the type `EventStruct` by replacing the final component `Top` with $\mathtt{E} \rightarrow \mathtt{Label} \times \mathtt{Top}$ to get the subtype

$$\begin{array}{l}\mathtt{TaggedEventStruct} == \\ \quad \mathtt{E}{:}\mathbb{U} \times \mathtt{M{:}MessageStruct} \times \mathtt{E}{\rightarrow}|\mathtt{M}| \times \mathtt{E}{\rightarrow}\mathtt{Label} \times \mathtt{E}{\rightarrow}\mathbb{B} \times \mathtt{E}{\rightarrow}\mathtt{Label} \times \mathtt{Top}\end{array}$$

If `E` is a tagged event structure, then it is also an event structure, but it has an additional component, a function $\mathtt{tag}_{\mathrm{E}}$ of type $|\mathtt{E}| \rightarrow \mathtt{Label}$. A trace `tr` over `E` is still a trace as defined previously, but every event `tr[i]` has a tag $\mathtt{tag}_{\mathrm{E}}(\mathtt{tr[i]})$. The sublist of `tr` consisting of all events in `tr` with a given tag `tg` is defined by

$$\mathtt{tr}|_{\mathtt{tg}} == \mathtt{<e} \in \mathtt{tr} \mid \mathtt{tag}_{\mathrm{E}}(\mathtt{e}) = \mathtt{tg>}$$

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

$$\begin{array}{l}\mathtt{Tag\text{-}by\text{-}msg}_{\mathrm{E}}(\mathtt{tr}) == \\ \quad \forall \mathtt{i,j}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\ \ (\mathtt{tr[i]} \ \mathtt{=msg=}_{\mathrm{E}}\ \mathtt{tr[j]}) \Rightarrow (\mathtt{tag}_{\mathrm{E}}(\mathtt{tr[i]}) = \mathtt{tag}_{\mathrm{E}}(\mathtt{tr[j]}))\end{array}$$

5 Fusion of Trace Properties

If the protocol associated with each label is guaranteeing some trace property `P`, and `tr` is a trace, then we will have $\forall \mathtt{m{:}Label}.\ \mathtt{P}(\mathtt{tr}|_{\mathrm{M}})$. A switch protocol must guarantee some additional property `I` that is strong enough to guarantee that property `P` holds of the whole trace `tr`. If this is the case for `I` and `P` then we say that `I` is a *fusion condition* for `P`, and we define

$$\mathtt{I\ fuses\ P} == \forall \mathtt{tr{:}Trace}_{\mathrm{E}}.\ (\forall \mathtt{m{:}Label}.\ \mathtt{P}(\mathtt{tr}|_{\mathrm{M}})) \Rightarrow \mathtt{I(tr)} \Rightarrow \mathtt{P(tr)}$$

We will be looking for properties `I` that are fusion conditions for whole classes of trace properties `P`. We will show that certain properties are fusion conditions for all properties `P` that satisfy certain meta-properties. We start this investigation with some simple lemmas about fusion.

Lemma 5.1

$$\begin{array}{l}\forall \mathtt{E{:}TaggedEventStruct}.\ \forall \mathtt{I,P,Q{:}TraceProperty}_{\mathrm{E}}. \\ \quad (\mathtt{I\ fuses\ P}) \Rightarrow (\mathtt{I\ fuses\ Q}) \Rightarrow (\mathtt{I\ fuses\ (P} \wedge \mathtt{Q)})\end{array}$$

Lemma 5.2

$$\begin{array}{l}\forall \mathtt{E{:}TaggedEventStruct}.\ \forall \mathtt{I,J,P{:}TraceProperty}_{\mathrm{E}}. \\ \quad (\mathtt{J} \rhd \mathtt{I}) \Rightarrow (\mathtt{I\ fuses\ P}) \Rightarrow (\mathtt{J\ fuses\ P})\end{array}$$

Lemma 5.3 (fusion simplification)

$\forall$`E:TaggedEventStruct.` $\forall$`I,J,P:TracePropertyE.`
`((I` $\wedge$ `J) fuses P)` $\Rightarrow$ `(I fuses J)` $\Rightarrow$ `(P` $\triangleright$ `J)` $\Rightarrow$ `(I fuses P)`

The proofs of these lemmas are straightforward. We will also need a few lemmas about fusion conditions for the properties $\texttt{Causal}_{\mathrm{E}}$ and $\texttt{No-dup-deliver}_{\mathrm{E}}$.

Lemma 5.4

$\forall\texttt{E:TaggedEventStruct. P}_{\texttt{True}}\texttt{ fuses Causal}_{\mathrm{E}}$

Proof: This lemma says that if $\forall$`m:Label.` $\texttt{Causal}_{\mathrm{E}}(\texttt{tr}|_{\mathrm{M}})$ then $\texttt{Causal}_{\mathrm{E}}(\texttt{tr})$. To show this, let `tr[j]` be a member of `tr`. We must show that there is an `i` $\leq$ `j` such that `tr[i]` is the send event for `tr[j]`. Event `tr[j]` has some tag `m` and using the causal property on $\texttt{tr}|_{\mathrm{M}}$, we find a send event `x` for `tr[j]` in $\texttt{tr}|_{\mathrm{M}}$. Since $\texttt{tr}|_{\mathrm{M}}$ is a sublist of `tr`, event `x` also precedes `j` in the trace `tr`. □

Lemma 5.5

$\forall\texttt{E:TaggedEventStruct. Tag-by-msg}_{\mathrm{E}}\texttt{ fuses No-dup-deliver}_{\mathrm{E}}$

Proof: If trace `tr` has a duplicate delivery of the message in event `x`, then trace $\texttt{tr}|_{\mathrm{M}}$, where $\texttt{m = tag}_{\mathrm{E}}\texttt{(x)}$, also has a duplicate delivery because, if `tr` has the $\texttt{Tag-by-msg}_{\mathrm{E}}$ property, all the relevant events have the same tag, `m`. □

Lemma 5.6

$\forall$`E:TaggedEventStruct.` $\forall$`tr:|E| List.`
$(\forall\texttt{m:Label. Causal}_{\mathrm{E}}(\texttt{tr}|_{\mathrm{M}}))\Rightarrow\texttt{No-dup-send}_{\mathrm{E}}\texttt{(tr)}\Rightarrow\texttt{Tag-by-msg}_{\mathrm{E}}\texttt{(tr)}$

Proof: If `x` is an event in `tr`, it has tag $\texttt{m = tag}_{\mathrm{E}}\texttt{(x)}$ and since $\texttt{tr}|_{\mathrm{M}}$ is causal, there is a send event for `x` with tag `m`. So any event `x` has a send event with the same tag. If `tr` has the $\texttt{No-dup-send}_{\mathrm{E}}$ property, then any two events with the same message must have the same send event and therefore the same tag. □

Lemma 5.7

$\forall$`E:TaggedEventStruct.` $\forall$`P,I:|E| List` $\rightarrow\mathbb{P}$.
$\texttt{(P}\triangleright\texttt{Causal}_{\mathrm{E}})\Rightarrow\texttt{((I}\wedge\texttt{No-dup-send}_{\mathrm{E}}\wedge\texttt{Tag-by-msg}_{\mathrm{E}}\texttt{) fuses P)}$
$\Rightarrow\texttt{((I}\wedge\texttt{No-dup-send}_{\mathrm{E}}\texttt{) fuses P)}$

Proof: This follows easily from lemma 5.6. □

Lemma 5.8

$\forall$`E:TaggedEventStruct.` $\forall$`P,I:|E| List` $\rightarrow\mathbb{P}$.
$\texttt{(P}\triangleright\texttt{(Causal}_{\mathrm{E}}\wedge\texttt{No-dup-deliver}_{\mathrm{E}}\texttt{))}$
$\Rightarrow\texttt{((I}\wedge\texttt{No-dup-send}_{\mathrm{E}}\wedge\texttt{Tag-by-msg}_{\mathrm{E}}\wedge\texttt{Causal}_{\mathrm{E}}\wedge\texttt{No-dup-deliver}_{\mathrm{E}}\texttt{) fuses P)}$
$\Rightarrow\texttt{((I}\wedge\texttt{No-dup-send}_{\mathrm{E}}\texttt{) fuses P)}$

Proof: By lemma 5.7, it is enough to show that $\texttt{(I}\wedge\texttt{No-dup-send}_{\mathrm{E}}\wedge\texttt{Tag-by-msg}_{\mathrm{E}}\texttt{) fuses P}$. Then using the fusion simplification lemma 5.3 with $\texttt{J = (Causal}_{\mathrm{E}}\wedge\texttt{No-dup-deliver}_{\mathrm{E}}\texttt{)}$, it is enough to show that

$$\texttt{(I}\wedge\texttt{No-dup-send}_{\mathrm{E}}\wedge\texttt{Tag-by-msg}_{\mathrm{E}}\texttt{) fuses (Causal}_{\mathrm{E}}\wedge\texttt{No-dup-deliver}_{\mathrm{E}}\texttt{)}$$

This follows from lemma 5.1, lemma 5.2, lemma 5.4, and lemma 5.5. □

5.1 Memoryless-Composable Induction

We are looking for a metaproperty $\texttt{switchable}_{\mathrm{E}}$ and a property $\texttt{switch\_inv}_{\mathrm{E}}$ such that $\texttt{switch\_inv}_{\mathrm{E}}$ is a fusion condition for any property P that satisfies $\texttt{switchable}_{\mathrm{E}}\texttt{(P)}$. We find this pair by a sequence of refinements.

We begin by assuming that P will be a memoryless, composable, safety property. So it satisfies the metaproperty

$$
\begin{aligned}
&\texttt{MCS}_{E}\texttt{(P) ==}\\
&\quad\quad \texttt{memorylessR}_{\mathrm{E}}\ \texttt{preserves P}\\
&\quad \wedge\ \texttt{(ternary) composableR}_{\mathrm{E}}\ \texttt{preserves P}\\
&\quad \wedge\ \texttt{safetyR}_{\mathrm{E}}\ \texttt{preserves P}
\end{aligned}
$$

We can now define a basic condition, *single-tag decomposable*, that fuses any MCS property.

$$
\begin{aligned}
&\texttt{single-tag-decomposable}_{\mathrm{E}}\texttt{(L) ==}\\
&\quad (\neg\texttt{(L = []))}\\
&\quad \Rightarrow\ (\exists \texttt{L}_1\texttt{,L}_2\texttt{:Trace}_{\mathrm{E}}.\\
&\quad\quad\quad\quad \texttt{(L = (L}_1\ \texttt{@ L}_2\texttt{))}\\
&\quad\quad\quad \wedge\ (\neg\texttt{(L}_2\ \texttt{= []))}\\
&\quad\quad\quad \wedge\ (\forall \texttt{x} \in \texttt{L}_1.\ \forall \texttt{y} \in \texttt{L}_2.\ \neg\texttt{(x =msg=}_{\mathrm{E}}\ \texttt{y))}\\
&\quad\quad\quad \wedge\ (\exists \texttt{m:Label}.\ \forall \texttt{x} \in \texttt{L}_2.\ \texttt{tag}_{\mathrm{E}}\texttt{(x) = m))}
\end{aligned}
$$

It says that any non null trace L can be decomposed as the append of two lists, $\texttt{L}_1$ and $\texttt{L}_2$, such that the two lists have no messages in common and the second list $\texttt{L}_2$ is non null and all events in it have the same tag. The MCS induction theorem states that a single-tag decomposable, safety property is a fusion condition for any MCS property.

Theorem 5.1 (MCS induction)

$$
\begin{aligned}
&\forall \texttt{E:TaggedEventStruct}.\ \forall \texttt{P,I:TraceProperty}_{\mathrm{E}}.\\
&\quad \texttt{MCS}_{E}\texttt{(P)}\\
&\quad \Rightarrow\ \texttt{safetyR}_{\mathrm{E}}\ \texttt{preserves I}\\
&\quad \Rightarrow\ \texttt{(I}\ \rhd\ \texttt{single-tag-decomposable}_{\mathrm{E}}\texttt{)}\\
&\quad \Rightarrow\ \texttt{(I fuses P)}
\end{aligned}
$$

Proof: We have to prove that $\texttt{I(tr)} \Rightarrow \texttt{P(tr)}$ under the assumptions that $\forall \texttt{m:Label}.\ \texttt{P(tr}|_{\mathrm{M}}\texttt{)}$ and the other hypotheses in the theorem. We proceed by induction on the length of tr.

base case: If tr has length 0, then it is the null list. Then, for any label m, $\texttt{tr}|_{\mathrm{M}} = \texttt{tr}$, so P(tr) by hypothesis.

induction step: Now tr is non null. Since I(tr), and since I refines single tag decomposability, we can find message-disjoint $\texttt{tr}_1$ and $\texttt{tr}_2$ such that $\texttt{tr = (tr}_1\ \texttt{@ tr}_2\texttt{)}$ and $\texttt{tr}_2$ is non null and has only one tag. Since the length of $\texttt{tr}_1$ is less than the length of tr, we can apply the induction hypothesis to conclude that $\texttt{P(tr}_1\texttt{)}$ provided that we can show that $\texttt{I(tr}_1\texttt{)}$ and $\forall \texttt{m:Label}.\ \texttt{P(tr}_1|_{\mathrm{M}}\texttt{)}$. The first of these follows from the assumption that I is a safety property, and the second follows from the assumption that P is a safety property and from the fact that $\forall \texttt{m:Label}.\ \texttt{tr}_1 \leq \texttt{tr} \Rightarrow \texttt{tr}_1|_{\mathrm{M}} \leq \texttt{tr}_1|_{\mathrm{M}}$. Since P is a composable property, we can conclude P(tr) if we can show $\texttt{P(tr}_2\texttt{)}$. By assumption, there is a tag m such that $\texttt{tr}_2 = \texttt{tr}_2|_{\mathrm{M}}$, so it's enough to show $\texttt{P(tr}_2|_{\mathrm{M}}\texttt{)}$. But $\texttt{tr}|_{\mathrm{M}} = \texttt{(tr}_1|_{\mathrm{M}}\ \texttt{@ tr}_2|_{\mathrm{M}}\texttt{)}$ and $\texttt{tr}_1$ and $\texttt{tr}_2$ have no messages in common, and therefore $\texttt{tr}_2|_{\mathrm{M}}$ can be obtained by removing all messages in $\texttt{tr}_1$ from $\texttt{tr}|_{\mathrm{M}}$. Since $\texttt{P(tr}|_{\mathrm{M}}\texttt{)}$ is true by assumption and since P is memoryless, we can conclude that $\texttt{P(tr}_2|_{\mathrm{M}}\texttt{)}$ and we are done. □

5.2 Switch Decomposability

Single-tag decomposabilty says that for a non null list of tagged events there must exist a decomposition into two suitable lists. We refine this property with a somewhat more constructive condition that we call *switch decomposability.* It says that for a non null list `L` of tagged events there must exist a decidable criterion `Q` on the times (the list indices) that satisfies a number of closure conditions. We will use the criterion `Q` to partition the list into two parts by defining the *message closure* of `Q`

$$\texttt{C(Q)(i) == } \exists \texttt{k:}\mathbb{N}_{||L||}\texttt{. Q(k)} \wedge \texttt{(L[k] =msg=}_{E}\texttt{ L[i])}$$

and then partitioniong `L` into `L-CQ`, containing those `L[i]` for which `C(Q)(i)`, and `L-notCQ`, the rest.

$$\begin{array}{l} \texttt{switch-decomposable}_{E}\texttt{(L) ==} \\ \quad \texttt{(L = [])} \\ \quad \vee\ (\exists \texttt{Q:}\mathbb{N}_{||L||} \to \mathbb{P}. \\ \qquad\qquad (\forall \texttt{i:}\mathbb{N}_{||L||}\texttt{. Dec(Q(i)))} \\ \qquad\quad \wedge\ (\exists \texttt{i:}\mathbb{N}_{||L||}\texttt{. Q(i))} \\ \qquad\quad \wedge\ (\forall \texttt{i:}\mathbb{N}_{||L||}\texttt{. Q(i)} \Rightarrow \texttt{is-send}_{E}\texttt{(L[i]))} \\ \qquad\quad \wedge\ (\forall \texttt{i,j:}\mathbb{N}_{||L||}\texttt{. Q(i)} \Rightarrow \texttt{Q(j)} \Rightarrow \texttt{(tag}_{E}\texttt{(L[i]) = tag}_{E}\texttt{(L[j])))} \\ \qquad\quad \wedge\ (\forall \texttt{i,j:}\mathbb{N}_{||L||}\texttt{. Q(i)} \Rightarrow \texttt{(i} \le \texttt{j)} \Rightarrow \texttt{C(Q)(j)))} \end{array}$$

We show that this property is a refinement of the single-tag decomposability in the following

Theorem 5.2

$$\begin{array}{l} \forall \texttt{E:TaggedEventStruct} \\ \qquad \texttt{(switch-decomposable}_{E} \wedge \texttt{Tag-by-msg}_{E} \wedge \texttt{Causal}_{E} \wedge \texttt{No-dup-send}_{E}\texttt{)} \\ \quad \rhd\ \texttt{single-tag-decomposable}_{E} \end{array}$$

Proof: We have a switch-decomposable list `L` that also satisfies the other three properties and, if `L` is non null we must produce a single-tag decomposition. In this case there is a `Q` with the properties given in the definition of switch-decomposable. The first of these properties says that `Q(i)` is decidable. From this, we can show that `C(Q)(i)` is also decidable, and therefore we can filter `L` on $\neg$`C(Q)` and `C(Q)` to get L_1 `= L-notCQ` and L_2 `= L-CQ`. We claim that this is a single-tag decomposition of `L`. We have to show that L_2 is non null, that L_2 is a final segment of `L`, that L_2 has only one tag, and that L_2 and L_1 have no messages in common. ¿From the second property of `Q` we see that L_2 is non null. By definition of the message closure, `C(Q)`, every event in `L` that has the same message as an event in L_2 must also be in L_2. Therefore L_2 and L_1 can have no messages in common. The fourth property of `Q` says that all events `L[i]` for which `Q(i)` holds, have the same tag. Everything in L_2 has the same message as one of these, so from the $\texttt{Tag-by-msg}_E$ property of `L`, we deduce that everything in L_2 has the same tag. To see that L_2 is a final segment of `L`, suppose that `C(Q)(i)` and $\texttt{i} \le \texttt{j}$. We must show that `C(Q)(j)`. Using the fifth property of `Q`, it's enough to find an $\texttt{i}'$ with $(\texttt{i}' \le \texttt{i}) \wedge \texttt{Q(i}'\texttt{)}$. But, by definition, $\texttt{C(Q)(i)} \Rightarrow (\exists \texttt{k:}\mathbb{N}_{||L||}\texttt{. Q(k)} \wedge \texttt{(L[k] =msg=}_{E}\texttt{ L[i]))}$. We are done if we show that any such `k` satisfies $\texttt{k} \le \texttt{i}$. This follows from the third property of `Q`, which implies that `L[k]` is a send event, and the two properties, $\texttt{No-dup-send}_E$ and $\texttt{Causal}_E$, which imply that for every event `L[i]` there is a unique send event with the same message, and that that send event must occur before `i`. ☐

5.3 The Switch Invariant

The switch guarantees that if two messages are sent using different protocols, then before the second message is delivered at any location, the first message must already have been delivered at that location. Thus, the switch guarantees the following invariant.

$$
\begin{array}{l}
\texttt{switch\_inv}_{\mathrm{E}}\texttt{(tr) ==} \\
\quad \forall \texttt{i,j,k}:\mathbb{N}_{\|\texttt{tr}\|}. \\
\qquad\quad \texttt{(i < j)} \\
\qquad \Rightarrow\ \texttt{is-send}_{\mathrm{E}}\texttt{(tr[i])} \\
\qquad \Rightarrow\ \texttt{is-send}_{\mathrm{E}}\texttt{(tr[j])} \\
\qquad \Rightarrow\ \texttt{(}\neg\texttt{(tag}_{\mathrm{E}}\texttt{(tr[i]) = tag}_{\mathrm{E}}\texttt{(tr[j])))} \\
\qquad \Rightarrow\ \texttt{tr[j] delivered at time k} \\
\qquad \Rightarrow\ \texttt{(}\exists \texttt{k}':\mathbb{N}_{\|\texttt{tr}\|}. \\
\qquad\qquad \texttt{(k}'\texttt{<k)} \ \wedge\ \texttt{tr[i] delivered at time k}' \ \wedge\ \texttt{(loc}_{\mathrm{E}}\texttt{(tr[k}'\texttt{]) = loc}_{\mathrm{E}}\texttt{(tr[k])))}
\end{array}
$$

We will show that certain strengthening of this invariant refines the switch-decomposablity property. This will be the crucial step in the proof of the main theorem. To accomplish this step, we first reformulate the switch invariant in a form that is more convienient for the argument. We now define a key relation $\texttt{switchR}_{\texttt{tr}}$. It relates two times `i` and `j` if the events at those times were send events and their messages were delivered out of order at some location. Note that $\texttt{switchR}_{\texttt{tr}}$ is a symmetric relation on the times $\mathbb{N}_{\|\texttt{tr}\|}$.

$$
\begin{array}{l}
\texttt{i switchR}_{\texttt{tr}}\texttt{ j ==} \\
\quad \texttt{is-send}_{\mathrm{E}}\texttt{(tr[i])} \\
\quad \wedge\ \texttt{is-send}_{\mathrm{E}}\texttt{(tr[j])} \\
\quad \wedge\ \texttt{(((i < j)} \ \wedge\ \texttt{tr[j] somewhere delivered before tr[i])} \\
\qquad \vee\ \texttt{((j < i)} \ \wedge\ \texttt{tr[i] somewhere delivered before tr[j]))}
\end{array}
$$

The switch invariant says that events whose times are related by $\texttt{switchR}_{\texttt{tr}}$ must have the same tag. So, using simple logic, we can prove

Lemma 5.9

$$
\begin{array}{l}
\forall \texttt{E:TaggedEventStruct}.\ \forall \texttt{tr:|E| List}. \\
\qquad \texttt{switch\_inv}_{\mathrm{E}}\texttt{(tr)} \\
\quad \Leftrightarrow\ \forall \texttt{i,j}:\mathbb{N}_{\|\texttt{tr}\|}.\ \ \texttt{(i switchR}_{\texttt{tr}}\texttt{ j)} \ \Rightarrow\ \texttt{(tag}_{\mathrm{E}}\texttt{(tr[i]) = tag}_{\mathrm{E}}\texttt{(tr[j]))}
\end{array}
$$

We want to show that a strengthening of $\texttt{switch\_inv}_{\mathrm{E}}$ refines $\texttt{switch-decomposable}_{\mathrm{E}}$. For a non null trace `tr` satisfying $\texttt{switch\_inv}_{\mathrm{E}}\texttt{(tr)}$ we must find the criterion `Q` on the times $\mathbb{N}_{\|\texttt{tr}\|}$ that satisfies the five properties in the definition of switch-decomposability. If `tr` also satisfies $\texttt{Causal}_{\mathrm{E}}\texttt{(tr)}$ then it must contain a send event and hence there must be a time `ls` which is the time of the last send event in `tr`. We will define `Q` to hold on all times related to `ls` by $\texttt{switchR}_{\texttt{tr}}$.

$$\texttt{Q(i) = (i (switchR}_{\texttt{tr}}{}^{*}\texttt{) ls)}$$

It is fairly easy to show that Q has the first four of the five required properties. First, Q is decidable because $\mathtt{switchR_{tr}}$ is decidable and the transitive closure of a decidable relation over a finite domain is also decidable. Second, Q is non empty beacause Q(ls) holds. The third requirement follows from the the fact that ls is the time of a send event and the following lemma, which is proved by induction on the transitive closure from the fact that $\mathtt{switchR_{tr}}$ relates only times of send events.

Lemma 5.10

$$
\begin{array}{l}
\forall \mathtt{E{:}EventStruct}.\ \forall \mathtt{tr{:}|E|\ List}.\ \forall \mathtt{ls,i}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\\
\quad \mathtt{is\text{-}send}_{\mathrm{E}}\mathtt{(tr[ls])} \Rightarrow \mathtt{(i\ (switchR_{tr}{}^{*})\ ls)} \Rightarrow \mathtt{is\text{-}send}_{\mathrm{E}}\mathtt{(tr[i])}
\end{array}
$$

Similarly, the fourth requirement, that the events of times satisfying Q all have the same tag, is a consequence of the following lemma, proved by induction from lemma 5.9.

Lemma 5.11

$$
\begin{array}{l}
\forall \mathtt{E{:}TaggedEventStruct}.\ \forall \mathtt{tr{:}|E|\ List}.\ \forall \mathtt{ls}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\\
\quad \mathtt{switch\_inv}_{\mathrm{E}}\mathtt{(tr)}\\
\quad \Rightarrow (\forall \mathtt{i,j}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\\
\qquad\quad \mathtt{(i\ (switchR_{tr}{}^{*})\ ls)}\\
\qquad \Rightarrow \mathtt{(j\ (switchR_{tr}{}^{*})\ ls)}\\
\qquad \Rightarrow (\mathtt{tag}_{\mathrm{E}}\mathtt{(tr[i])} = \mathtt{tag}_{\mathrm{E}}\mathtt{(tr[j])}))
\end{array}
$$

It remains only to establish the fifth property of Q, namely,

$$
\forall \mathtt{i,j}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\ \ \mathtt{Q(i)} \Rightarrow (\mathtt{i} \leq \mathtt{j}) \Rightarrow \mathtt{C(Q)(j)}
$$

To prove this property of Q we will need a stronger invariant than the property $\mathtt{switch\_inv}_{\mathrm{E}} \wedge \mathtt{Causal}_{\mathrm{E}}$ used so far. Without strengthening the invariant we can prove a weaker version of the requirement. We can prove

$$
\forall \mathtt{i,j}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\ \ \mathtt{Q(i)} \Rightarrow (\mathtt{i} \leq \mathtt{j}) \Rightarrow \mathtt{is\text{-}send}_{\mathrm{E}}\mathtt{(tr[j])} \Rightarrow \mathtt{Q(j)}
$$

This establishes the requirement for the case when the event at time j is a send event, and we need this lemma to prove the full requirement. For this lemma, the only assumption we need is that ls is the time of the last send event.

Lemma 5.12

$$
\begin{array}{l}
\forall \mathtt{E{:}EventStruct}.\ \forall \mathtt{tr{:}|E|\ List}.\ \forall \mathtt{ls}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\\
\quad \mathtt{is\text{-}send}_{\mathrm{E}}\mathtt{(tr[ls])}\\
\quad \Rightarrow (\forall \mathtt{j}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\ (\mathtt{ls} < \mathtt{j}) \Rightarrow (\neg \mathtt{is\text{-}send}_{\mathrm{E}}\mathtt{(tr[j])}))\\
\quad \Rightarrow (\forall \mathtt{i,j}{:}\mathbb{N}_{\|\mathtt{tr}\|}.\\
\qquad\quad (\mathtt{i} \leq \mathtt{j})\\
\qquad \Rightarrow \mathtt{is\text{-}send}_{\mathrm{E}}\mathtt{(tr[j])}\\
\qquad \Rightarrow \mathtt{(i\ (switchR_{tr}{}^{*})\ ls)}\\
\qquad \Rightarrow \mathtt{(j\ (switchR_{tr}{}^{*})\ ls))}
\end{array}
$$

Proof: We assume that ls is the time of the last send, that $\mathtt{i} \leq \mathtt{j}$, that $\mathtt{is\text{-}send}_{\mathrm{E}}(\mathtt{tr[j]})$, and that i $(\mathtt{switchR_{tr}}^*)$ ls. So, for some n, i $\mathtt{switchR_{tr}}^{\mathtt{n}}$ ls, and we prove by induction on n that j $(\mathtt{switchR_{tr}}^*)$ ls.

base case: When n = 0, then i = ls, so $\mathtt{ls} \leq \mathtt{j}$. Since ls is the last send, we have j = ls and hence, j $(\mathtt{switchR_{tr}}^*)$ ls.

induction step: Now 0 < n, and i $\mathtt{switchR_{tr}}^{\mathtt{n}}$ ls, so by definition, there is a time z such that i $\mathtt{switchR_{tr}}$ z and z $\mathtt{switchR_{tr}}^{\mathtt{n-1}}$ ls. If $\mathtt{z} \leq \mathtt{j}$, then by induction, j $(\mathtt{switchR_{tr}}^*)$ ls. So, we may assume j < z. Also, if j = i, then we have j $(\mathtt{switchR_{tr}}^*)$ ls, by hypothesis. So, we may assume i < j. So we also have i < z, and since i $\mathtt{switchR_{tr}}$ z, we must have tr[z] somewhere delivered before tr[i]. Then by lemma 4.1, we have tr[z] somewhere delivered before tr[j] $\vee$ tr[j] somewhere delivered before tr[i]. In the first case, because j < z we have j $\mathtt{switchR_{tr}}$ z and hence j $\mathtt{switchR_{x}}^{\mathtt{n}}$ ls. In the second case, because i < j we have j $\mathtt{switchR_{tr}}$ i and hence j $\mathtt{switchR_{tr}}^{\mathtt{n+1}}$ ls. So, in either case, j $(\mathtt{switchR_{tr}}^*)$ ls and we are done. □

To prove the complete closure requirement, $\forall \mathtt{i},\mathtt{j}:\mathbb{N}_{\|\mathtt{tr}\|}.\ \mathtt{Q(i)} \Rightarrow (\mathtt{i} \leq \mathtt{j}) \Rightarrow \mathtt{C(Q)(j)}$, we will have to assume that the trace tr has a certain normal form. We will be able to put a trace into normal form using only operations that preserve the relation $\mathtt{asyncR}_{\mathrm{E}} \vee \mathtt{delayableR}_{\mathrm{E}}$. So, for asynchronous, delayable properties, P, if P holds of the normal form of trace tr, then it also holds of tr.

We put a trace into normal form by moving send events as late as possible and by reordering asynchronous deliver events to match the order of their send events. More constructively, we apply the following normalization algorithm. If we find an adjacent pair of events (a,b) in the trace where a is a send event and b is a deliver event, then, unless they contain the same message, we swap the two events so that the send event a occurs later than the deliver event b. If we find an adjacent pair of deliver events (a,b) for which there are corresponding send events that occur in the opposite order, then if events a and b are at different locations, we swap them so that their order is the same as their corresponding send events. Under some additional assumptions, we will prove that this algorithm terminates and results in a trace that satisfies the following normal form property, that we call *asynchronous-delayable normal form* or *AD-normal*.

$$
\begin{array}{l}
\mathtt{AD\text{-}normal}_{\mathrm{E}}(\mathtt{tr}) == \\
\quad \forall \mathtt{i}:\mathbb{N}_{\|\mathtt{tr}\|-1} \\
\qquad (\mathtt{is\text{-}send}_{\mathrm{E}}(\mathtt{tr[i]}) \Rightarrow (\neg\mathtt{is\text{-}send}_{\mathrm{E}}(\mathtt{tr[i+1]})) \Rightarrow (\mathtt{tr[i]} =\!\mathtt{msg}\!=_{\mathrm{E}} \mathtt{tr[i+1]})) \\
\qquad \wedge\ ((\exists \mathtt{x},\mathtt{y}:\mathbb{N}_{\|\mathtt{tr}\|}. \\
\qquad\qquad (\mathtt{x} < \mathtt{y}) \\
\qquad\qquad \wedge\ \mathtt{is\text{-}send}_{\mathrm{E}}(\mathtt{tr[x]}) \\
\qquad\qquad \wedge\ \mathtt{is\text{-}send}_{\mathrm{E}}(\mathtt{tr[y]}) \\
\qquad\qquad \wedge\ \mathtt{tr[x]\ delivered\ at\ time\ i+1} \\
\qquad\qquad \wedge\ \mathtt{tr[y]\ delivered\ at\ time\ i)} \\
\qquad\ \Rightarrow (\mathtt{loc}_{\mathrm{E}}(\mathtt{tr[i]}) = \mathtt{loc}_{\mathrm{E}}(\mathtt{tr[i+1]})))
\end{array}
$$

Assuming that trace tr satisfies $\mathtt{AD\text{-}normal}_{\mathrm{E}}$ and two additional properties, $\mathtt{No\text{-}dup\text{-}deliver}_{\mathrm{E}}$ and $\mathtt{Causal}_{\mathrm{E}}$, and the assumptions needed for lemma 5.12, we can now prove

Lemma 5.13

$$\forall \mathtt{i,j}:\mathbb{N}_{||\mathtt{tr}||}.\;\; \mathtt{Q(i)} \;\Rightarrow\; \mathtt{(i} \le \mathtt{j)} \;\Rightarrow\; \mathtt{C(Q)(j)}$$

Proof: We proceed by induction on `j - i`.

base case: If `(j - i) = 0`, then `j = i`, so `Q(j)`, and hence `C(Q)(j)`.

induction step: If `(j - i) > 0`, then $\mathtt{i} \le \mathtt{(j - 1)}$. We assume `Q(i)`, and this implies that `tr[i]` is a send event. We must show `C(Q)(j)`. If the event at time `j` is a send event, then lemma 5.12 implies that `Q(j)`. So we may assume that `tr[j]` is a deliver event.

case 1 `tr[j-1]` is a send event. Lemma 5.12 implies that `Q(j - 1)` and the normal form implies that $\mathtt{(tr[j\text{-}1]\ {=}msg{=}_{E}\ tr[j])}$. Hence `C(Q)(j)`.

case 2 `tr[j-1]` is a deliver event. By induction, we have `C(Q)(j - 1)`, so there is a time `k` such that `Q(k)` and the events at times `k` and `j - 1` have the same message. So the event at time `k` is the send event for the deliver event at time `j - 1`. By the $\mathtt{Causal}_{E}$ property, there is also a time $\mathtt{j}'$ when the send event for the delivery at time `j` occured. We claim that $\mathtt{Q(j')}$ holds, and hence `C(Q)(j)`.

case 2.1 $\mathtt{j}' < \mathtt{k}$. In this case, the deliveries at times `j - 1` and `j` are out of order, so by the normal form property, they must have the same location, call it `p`. Since we have the $\mathtt{No\text{-}dup\text{-}deliver}_{E}$ property, there cannot be any delivery of the message sent at time $\mathtt{j}'$ to location `p` other than the one at time `j`. Therefore we have $\mathtt{tr[k]\ somewhere\ delivered\ before\ tr[j']}$, and this implies that $\mathtt{j}'$ $\mathtt{switchR}_{\mathtt{tr}}$ `k`. Since `Q(k)`, and since `Q` is closed under $\mathtt{switchR}_{\mathtt{tr}}$, we get $\mathtt{Q(j')}$.

case 2.2 $\mathtt{k} \le \mathtt{j}'$. In this case, since the event at time $\mathtt{j}'$ is a send and since `Q(k)`, lemma 5.12 implies that $\mathtt{Q(j')}$. □

Putting together everything we have proved in this section, we see that we have proved the following

Theorem 5.3

$$\begin{array}{l}\forall \mathtt{E:TaggedEventStruct}\\ \quad (\mathtt{switch\_inv}_{E} \wedge \mathtt{Causal}_{E} \wedge \mathtt{AD\text{-}normal}_{E} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{E})\\ \quad \rhd\ \mathtt{switch\text{-}decomposable}_{E}\end{array}$$

The next theorem combines all the previous lemmas.

Theorem 5.4

$$\begin{array}{l}\forall \mathtt{E:TaggedEventStruct}.\ \forall \mathtt{P:TraceProperty}_{E}.\\ \quad \mathtt{MCS}_{E}\mathtt{(P)}\\ \Rightarrow\ (\mathtt{P} \rhd (\mathtt{Causal}_{E} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{E}))\\ \Rightarrow\ (((\mathtt{switch\_inv}_{E} \wedge \mathtt{AD\text{-}normal}_{E}) \wedge \mathtt{No\text{-}dup\text{-}send}_{E})\ \mathtt{fuses\ P})\end{array}$$

Proof: Using lemma 5.8, it is enough to show that `I fuses P` where

$$\begin{array}{l}\mathtt{I} = (\ (\mathtt{switch\_inv}_{E} \wedge \mathtt{AD\text{-}normal}_{E}) \wedge \mathtt{No\text{-}dup\text{-}send}_{E}\\ \qquad \wedge\ (\mathtt{Tag\text{-}by\text{-}msg}_{E} \wedge \mathtt{Causal}_{E} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{E}))\end{array}$$

We prove this using MCS-induction. The hypotheses of the MCS-induction require us to show that `I` is a safety property and that `I` is single-tag-decomposable. It is straightforward to show that each of the six conjoined properties in `I` is a safety property, and therefore `I` is a safety property. By theorem 5.2, to show that `I` is single-tag decomposable, it is enough to show that `I` is switch-decomposable, since `I` refines $\mathtt{Tag\text{-}by\text{-}msg}_{E}$, $\mathtt{Causal}_{E}$, and $\mathtt{No\text{-}dup\text{-}send}_{E}$. But `I` refines

$$\mathtt{switch\_inv}_{E} \wedge \mathtt{Causal}_{E} \wedge \mathtt{AD\text{-}normal}_{E} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{E}$$

and, by theorem 5.3, that property refines the switch-decomposabilty property. □

5.4 Removing the Normal Form Requirement

The conclusion of theorem 5.4 has the form `((I ∧ J) ∧ K) fuses P`, where `J` is the normal form property $\texttt{AD-normal}_{\mathrm{E}}$. We show that if `P` is asynchronous and delayable, then we can remove this requirement and prove `(I ∧ K) fuses P`. An asynchronous, delayable property `P` is preserved by the following relation

$$\texttt{adR}_{\mathrm{E}} \texttt{ == (delayableR}_{\mathrm{E}} \vee \texttt{asyncR}_{\mathrm{E}}\texttt{)}^{*}$$

This relation is a symmetric relation and also has the property that we call *tag-splitable.* If two traces are related by $\texttt{adR}_{\mathrm{E}}$, then so are their tagged subtraces for any given tag.

$$\begin{array}{l}\texttt{tag\_splitable}_{\mathrm{E}}\texttt{(R) ==}\\ \quad \forall \texttt{tr}_1\texttt{,tr}_2\texttt{:Trace}_{\mathrm{E}}.\ \ (\texttt{tr}_1 \texttt{ R } \texttt{tr}_2) \Rightarrow (\forall \texttt{m:Label. } \texttt{tr}_1|_{\mathrm{M}} \texttt{ R } \texttt{tr}_2|_{\mathrm{M}})\end{array}$$

Lemma 5.14

$$\forall \texttt{E:TaggedEventStruct. tag\_splitable}_{\mathrm{E}}(\texttt{adR}_{\mathrm{E}})$$

Proof: The reason that $\texttt{adR}_{\mathrm{E}}$ is tag-splitable is that it is the transitive closure of relations defined by swapping adjacent elements. If $\texttt{tr}_1$ is obtained from $\texttt{tr}_2$ by swapping adjacent elements, then $\texttt{tr}_1|_{\mathrm{M}}$ will either be equal to $\texttt{tr}_2|_{\mathrm{M}}$ (if the two elements swapped do not both have tag `m`) or else obtained by swapping adjacent elements of $\texttt{tr}_2|_{\mathrm{M}}$. □

The outline of the argument we use to remove the normal form requirement is contained in the following lemma.

Lemma 5.15

$$\begin{array}{l}\forall \texttt{E:TaggedEventStruct. } \forall \texttt{P,I,J,K:TraceProperty}_{\mathrm{E}}.\ \forall \texttt{R:Trace}_{\mathrm{E}} \rightarrow \texttt{Trace}_{\mathrm{E}} \rightarrow \mathbb{P}.\\ \qquad \texttt{tag\_splitable}_{\mathrm{E}}\texttt{(R)}\\ \quad \Rightarrow (\forall \texttt{tr}_1\texttt{,tr}_2\texttt{:Trace}_{\mathrm{E}}.\ \ (\texttt{tr}_1 \texttt{ R } \texttt{tr}_2) \Rightarrow (\texttt{tr}_2 \texttt{ R } \texttt{tr}_1))\\ \quad \Rightarrow \texttt{R preserves P}\\ \quad \Rightarrow \texttt{R preserves K}\\ \quad \Rightarrow (\forall \texttt{tr:Trace}_{\mathrm{E}}.\ (\texttt{I} \wedge \texttt{K})(\texttt{tr}) \Rightarrow (\exists \texttt{tr}'\texttt{:Trace}_{\mathrm{E}}.\ \texttt{I}(\texttt{tr}') \wedge \texttt{J}(\texttt{tr}') \wedge (\texttt{tr R tr}')))\\ \quad \Rightarrow (((\texttt{I} \wedge \texttt{J}) \wedge \texttt{K}) \texttt{ fuses P})\\ \quad \Rightarrow ((\texttt{I} \wedge \texttt{K}) \texttt{ fuses P})\end{array}$$

Proof: If `tr` has properties `I` and `K`, and if $\forall\texttt{m:Label. P(tr}|_{\mathrm{M}}\texttt{)}$, then we must show `P(tr)` assuming the six hypotheses. Using the fifth hypothesis, we find `tr′`, related to `tr` by `R` and satisfying `I` and `J`. Since `R preserves K`, trace `tr′` also satisfies `K`. Using the fusion hypothesis, we can conclude that `P(tr′)` if we can show $\forall\texttt{m:Label P(tr}'|_{\mathrm{M}}\texttt{)}$. This follows from the assumptions that `R` is tag-splitable and preserves `P`. Once we have `P(tr′)`, we conclude `P(tr)` from the assumptions that `R` is a symmetric relation and preserves `P`. □

The `K` we need in the previous argument is $\texttt{No-dup-send}_{\mathrm{E}}$, and the `R` is $\texttt{adR}_{\mathrm{E}}$. That `R preserves K` follows from the next two lemmas whose proofs are straightforward.

Lemma 5.16

$\forall$E:EventStruct. asyncR$_{\mathrm{E}}$ preserves No-dup-send$_{\mathrm{E}}$

Lemma 5.17

$\forall$E:EventStruct. delayableR$_{\mathrm{E}}$ preserves No-dup-send$_{\mathrm{E}}$

When we instantiate lemma 5.15 with K and R as above and with switch_inv$_{\mathrm{E}}$ for I and AD-normal$_{\mathrm{E}}$ for J, then the last hypothesis is theorem 5.4, and, as we have seen, the first four hypotheses are all satisfied. The fifth hypothesis becomes the following lemma, which we will prove.

Lemma 5.18

$\forall$E:TaggedEventStruct. $\forall$tr:Trace$_{\mathrm{E}}$.
(switch_inv$_{\mathrm{E}}$ $\wedge$ No-dup-send$_{\mathrm{E}}$)(tr)
$\Rightarrow$ ($\exists$tr$'$:Trace$_{\mathrm{E}}$. switch_inv$_{\mathrm{E}}$(tr$'$) $\wedge$ AD-normal$_{\mathrm{E}}$(tr$'$) $\wedge$ (tr adR$_{\mathrm{E}}$ tr$'$))

Once lemma 5.18 is proved, we see that we have the following theorem, which is essentially our main result.

Theorem 5.5

$\forall$E:TaggedEventStruct. $\forall$P:TraceProperty$_{\mathrm{E}}$.
MCS$_{E}$(P)
$\Rightarrow$ asyncR$_{\mathrm{E}}$ preserves P
$\Rightarrow$ delayableR$_{\mathrm{E}}$ preserves P
$\Rightarrow$ (P $\rhd$ (Causal$_{\mathrm{E}}$ $\wedge$ No-dup-deliver$_{\mathrm{E}}$))
$\Rightarrow$ ((switch_inv$_{\mathrm{E}}$ $\wedge$ No-dup-send$_{\mathrm{E}}$) fuses P)

Proof of lemma 5.18: We use the following general theorem on the existence of partially sorted lists. It says that by swapping adjacent elements x,y of list L, for which $\neg$P(x,y), provided that each swap results in a pair that satisfies P, we may reach a sorted list L$'$ in which all adjacent pairs satisfy P.

Theorem 5.6

$\forall$T:$\mathbb{U}$. $\forall$L:T List. $\forall$P:T $\to$ T $\to$ $\mathbb{P}$.
($\forall$x,y:T. Dec(P(x,y)))
$\Rightarrow$ ($\forall$x,y:T. ($\neg$P(x,y)) $\Rightarrow$ P(y,x))
$\Rightarrow$ ($\exists$L$'$:T List.
(L (swap-adjacent$_{(\neg P(x,y))}$*) L$'$) $\wedge$ ($\forall$i:$\mathbb{N}_{||L'||-1}$. P(L$'$[i],L$'$[i+1])))

Proof: The proof is by induction on the cardinality of the set of *bad* pairs – the pairs for which $\neg$P(x,y) and where x precedes y in L (but is not necessarily adjacent). If P is decidable, we can find an adjacent bad pair, if one exists, and show that swapping decreases the cardinality of the set of bad pairs. □

We apply theorem 5.6 with P instantitated to

```
ad-normalR_tr(a,b) ==
  (is-send_E(a) ⇒ is-deliver_E(b) ⇒ (a =msg=_E b))
  ∧ (is-deliver_E(a)
    ⇒ is-deliver_E(b)
    ⇒ (∃x,y:ℕ_||tr||.
          (x < y)
        ∧ is-send_E(tr[x])
        ∧ is-send_E(tr[y]) ∧ (tr[x] =msg=_E b)
        ∧ (tr[y] =msg=_E a)
     )
    ⇒ (loc_E(a) = loc_E(b)))
```

We must show that $\texttt{ad-normalR}_{\texttt{tr}}$ is decidable, which is easy, and that

```
(¬ad-normalR_tr(a,b)) ⇒ ad-normalR_tr(b,a)
```

This follows from the $\texttt{No-dup-send}_{\mathrm{E}}$ property of tr, since if (a,b) is a pair of out of order deliveries then (b,a) will not be out of order since there are unique send events for them. The partial sort theorem then gives us a trace tr′ reachable from trace tr by swapping adjacent pairs for which $\neg\texttt{ad-normalR}_{\texttt{tr}}\texttt{(a,b)}$, and for which all adjacent pair satisfy $\texttt{ad-normalR}_{\texttt{tr}}$. We must show that $\texttt{tr adR}_{\mathrm{E}}\texttt{ tr}'$, that $\texttt{switch\_inv}_{\mathrm{E}}\texttt{(tr}'\texttt{)}$, and that $\texttt{AD-normal}_{\mathrm{E}}\texttt{(tr}'\texttt{)}$. The first of these follows from the observation that swapping adjacent pairs for which $\neg\texttt{ad-normalR}_{\texttt{tr}}\texttt{(a,b)}$ always swaps pairs allowed by either the relation $\texttt{asyncR}_{\mathrm{E}}$ or the relation $\texttt{delayableR}_{\mathrm{E}}$. The second requirement follows from the assumption that $\texttt{switch\_inv}_{\mathrm{E}}\texttt{(tr)}$, and the following lemma, whose proof is straightforward, which shows that swapping pairs for which $\neg\texttt{ad-normalR}_{\texttt{tr}}\texttt{(a,b)}$ preserves the property $\texttt{switch\_inv}_{\mathrm{E}}$.

Lemma 5.19

```
∀E:TaggedEventStruct. ∀x:|E| List. ∀i:ℕ_||x||-1.
  switch_inv_E(x)
  ⇒ (¬is-send_E(x[i+1]))
  ⇒ (is-send_E(x[i]) ∨ (¬(loc_E(x[i]) = loc_E(x[i+1]))))
  ⇒ switch_inv_E(swap(x;i;i + 1))
```

To finish the proof of lemma 5.18 it remains to show that if all adjacent element of tr′ satisfy $\texttt{ad-normalR}_{\texttt{tr}}$ then tr′ satisfies $\texttt{AD-normal}_{\mathrm{E}}$. It can easily be seen that if all adjacent pairs satisfy $\texttt{ad-normalR}_{\texttt{tr}'}$ then tr′ is in normal form. But the dependence of $\texttt{ad-normalR}_{\texttt{tr}}$ on parameter tr is only on the order of send events in tr, and this order is preserved by the swaps that lead from tr to tr′. □

We restate theorem 5.5 by combining its hypotheses into the definition of a *switchable* property.

$$
\begin{array}{l}
\mathtt{switchable}_{\mathrm{E}}\mathtt{(P)} \ \mathtt{==} \\
\quad \mathtt{safetyR}_{\mathrm{E}} \ \mathtt{preserves\ P} \\
\quad \wedge \ \mathtt{memorylessR}_{\mathrm{E}} \ \mathtt{preserves\ P} \\
\quad \wedge \ \mathtt{(ternary)\ composableR}_{\mathrm{E}} \ \mathtt{preserves\ P} \\
\quad \wedge \ \mathtt{send\text{-}enabledR}_{\mathrm{E}} \ \mathtt{preserves\ P} \\
\quad \wedge \ \mathtt{asyncR}_{\mathrm{E}} \ \mathtt{preserves\ P} \\
\quad \wedge \ \mathtt{delayableR}_{\mathrm{E}} \ \mathtt{preserves\ P} \\
\quad \wedge \ \mathtt{(P} \rhd \mathtt{Causal}_{\mathrm{E}}\mathtt{)} \\
\quad \wedge \ \mathtt{(P} \rhd \mathtt{No\text{-}dup\text{-}deliver}_{\mathrm{E}}\mathtt{)}
\end{array}
$$

Theorem 5.7

$$
\begin{array}{l}
\forall \mathtt{E:TaggedEventStruct.} \ \forall \mathtt{P:TraceProperty}_{\mathrm{E}}. \\
\quad \mathtt{switchable}_{\mathrm{E}}\mathtt{(P)} \ \Rightarrow \ \mathtt{((switch\_inv}_{\mathrm{E}} \wedge \mathtt{No\text{-}dup\text{-}send}_{\mathrm{E}}\mathtt{)\ fuses\ P)}
\end{array}
$$

□

6 Proof of the Switch Theorem

We said that theorem 5.7 is essentially our main result, but it is a theorem about trace properties over a tagged event structure, and the fusion condition that is its conclusion is a condition on a single tagged trace. Our real main result will be about properties over an event structure, and the hypotheses will relate two traces, the traces "above" and "below" our switching protocol layer. The upper trace $\mathtt{tr}_u$ will be a trace over an event structure `E`, so it is a list of send and deliver events. The lower trace $\mathtt{tr}_l$ will be a list of some actions of a type `A`. These actions represent the channels between the switch layer and the different protocols being switched. Each of these actions is essentially a pair of an event from `|E|` and a label. That is, from an action $\mathtt{a} \in \mathtt{A}$ we can extract its event and a label identifying which protocol it is for. So we have functions $\mathtt{evt} \in \mathtt{A} \rightarrow \mathtt{|E|}$ and $\mathtt{tg} \in \mathtt{A} \rightarrow \mathtt{Label}$. From these functions, we can form a tagged event structure over `A` as follows:

$$
\mathtt{<A,evt,tg>}_{\mathrm{E}} \ \mathtt{==} \ \mathtt{<A,\ MS}_{\mathrm{E}}\mathtt{,\ msg}_{\mathrm{E}} \ \mathtt{o\ evt,\ loc}_{\mathrm{E}} \ \mathtt{o\ evt,\ is\text{-}send}_{\mathrm{E}} \ \mathtt{o\ evt,\ tg,\ \cdot>}
$$

If $\mathtt{tr}_l$ is a list of actions in `A`, then $\mathtt{map(evt;tr}_l\mathtt{)}$ is a list of events in `|E|`, i.e a trace over `E`. Thus, if `P` is a trace property over `E` then we define

$$
\mathtt{P}_{\mathtt{evt}}\mathtt{(L)} \ \mathtt{==} \ \mathtt{P(map(evt;L))}
$$

and $\mathtt{P}_{\mathtt{evt}}$ is a trace property over the induced tagged event structure $\mathtt{<A,evt,tg>}_{\mathrm{E}}$. Using these definitions and theorem 5.7 we get:

Theorem 6.1

$$
\begin{array}{l}
\forall \mathtt{E:EventStruct.} \ \forall \mathtt{P:|E|\ List} \rightarrow \mathbb{P}. \ \forall \mathtt{A:}\mathbb{U}. \ \forall \mathtt{evt:A} \rightarrow \mathtt{|E|}. \\
\quad \forall \mathtt{tg:A} \rightarrow \mathtt{Label.} \ \forall \mathtt{tr:A\ List.} \\
\quad \mathtt{switchable}_{\mathrm{E}}\mathtt{(P)} \\
\quad \Rightarrow \ \mathtt{No\text{-}dup\text{-}send}_{\mathrm{E}}\mathtt{(map(evt;tr))} \\
\quad \Rightarrow \ \mathtt{switch\_inv}_{\mathtt{<A,evt,tg>}_{\mathrm{E}}}\mathtt{(tr)} \\
\quad \Rightarrow \ \mathtt{(}\forall \mathtt{m:Label.\ P(map(evt;tr|}_{\mathrm{M}}\mathtt{)))} \\
\quad \Rightarrow \ \mathtt{P(map(evt;tr))}
\end{array}
$$

Proof: This follows from theorem 5.7 and the definition of fusion, once we establish that
$\texttt{switchable}_{\mathrm{E}}\texttt{(P)} \Rightarrow \texttt{switchable}_{\texttt{<A,evt,tg>}_{\mathrm{E}}}\texttt{(P}_{\texttt{evt}}\texttt{)}$ and
$\texttt{No-dup-send}_{\mathrm{E}}\texttt{(map(evt;tr}_l\texttt{))} \Rightarrow \texttt{No-dup-send}_{\texttt{<A,evt,tg>}_{\mathrm{E}}}\texttt{(tr}_l\texttt{)}$.
Both of these follow easily from the facts that the map operation commutes with every list operation used in the definitions of $\texttt{switchable}_{\mathrm{E}}$ and $\texttt{No-dup-send}_{\mathrm{E}}$, and that, by definition, `evt`, maps the event structure of `A` to `E`. □

The switching protocol layer delays the delivery of messages from one sub-protocol until the messages from the previous sub-protocol have been delivered. It thus reorders the trace of tagged events (the actions `A`) in its lower trace $\texttt{tr}_l$ so that the reordered trace, which we call $\texttt{tr}_m$, for the middle trace, satisfies the $\texttt{switch\_inv}_{\texttt{<A,evt,tg>}_{\mathrm{E}}}$ property. In this reordering, the switch does not change the ordering of actions from the same protocol, it only changes the ordering of actions from different protocols. Thus, the lower and middle traces, $\texttt{tr}_l$ and $\texttt{tr}_m$, will be related by the following *tag relation*

$$\texttt{R}_{\texttt{tg}} \texttt{ == swap-adjacent}_{(\neg(\texttt{tg(x) = tg(y)}))}{}^{*}$$

With the tags removed, the reordered actions in $\texttt{tr}_m$ are the same as the events in the upper trace $\texttt{tr}_u$, except that further delays in the switch may change the order of these events, and the upper trace $\texttt{tr}_u$ may contain additional send events that have not yet been communicated through the switch to the lower (or middle) trace. Thus, $\texttt{map(evt;tr}_m\texttt{)}$ is related to $\texttt{tr}_u$ by the *layer relation*:

$$\texttt{layerR}_{\mathrm{E}} \texttt{ == ((asyncR}_{\mathrm{E}} \vee \texttt{delayableR}_{\mathrm{E}}\texttt{)} \vee \texttt{send-enabledR}_{\mathrm{E}}\texttt{)}^{*}$$

Putting all of these things together, we have the following definition of the full invariant that the switch layer must guarantee.

$$\begin{array}{l}
\texttt{full_switch_inv(E;A;evt;tg;tr}_u\texttt{;tr}_l\texttt{) ==} \\
\quad \exists \texttt{tr}_m\texttt{:A List. (tr}_l \ \texttt{R}_{\texttt{tg}} \ \texttt{tr}_m\texttt{)} \\
\qquad\qquad \wedge \ \texttt{(map(evt;tr}_m\texttt{) layerR}_{\mathrm{E}} \ \texttt{tr}_u\texttt{)} \\
\qquad\qquad \wedge \ \texttt{switch_inv}_{\texttt{<A,evt,tg>}_{\mathrm{E}}}\texttt{(tr}_m\texttt{)}
\end{array}$$

We can finally derive our main theorem.

Theorem 6.2 (Main Switch Theorem)

$$\begin{array}{l}
\forall \texttt{E:EventStruct. } \forall \texttt{P:TraceProperty}_{\mathrm{E}}\texttt{. } \forall \texttt{A:}\mathbb{U}\texttt{. } \forall \texttt{evt:A} \rightarrow \texttt{|E|.} \\
\quad \forall \texttt{tg:A} \rightarrow \texttt{Label. } \forall \texttt{tr}_u\texttt{:Trace}_{\mathrm{E}}\texttt{. } \forall \texttt{tr}_l\texttt{:A List.} \\
\qquad \texttt{switchable}_{\mathrm{E}}\texttt{(P)} \\
\quad \Rightarrow \ \texttt{No-dup-send}_{\mathrm{E}}\texttt{(tr}_u\texttt{)} \\
\quad \Rightarrow \ \texttt{full_switch_inv(E;A;evt;tg;tr}_u\texttt{;tr}_l\texttt{)} \\
\quad \Rightarrow \ \texttt{(}\forall \texttt{m:Label. P(map(evt;tr}_l|_{\mathrm{M}}\texttt{)))} \\
\quad \Rightarrow \ \texttt{P(tr}_u\texttt{)}
\end{array}$$

Proof: We use theorem 6.1 with $\mathtt{tr}_m$ for tr. We must show

$\mathtt{No\text{-}dup\text{-}send}_{\mathrm{E}}(\mathtt{map(evt;tr}_m\mathtt{))}$

which follows from the assumption $\mathtt{No\text{-}dup\text{-}send}_{\mathrm{E}}(\mathtt{tr}_u)$ and the easy lemma that $\mathtt{No\text{-}dup\text{-}send}_{\mathrm{E}}$ is preserved by $\mathtt{layerR}_{\mathrm{E}}$. We must also show that

$\forall\mathtt{m:Label.}\ \mathtt{P(map(evt;tr}_m|_{\mathrm{M}}\mathtt{))}$

which follows from the assumption $\forall\mathtt{m:Label.}\ \mathtt{P(map(evt;tr}_l|_{\mathrm{M}}\mathtt{))}$ and the easy lemma that

$\mathtt{(tr1\ R_{tg}\ tr2)} \Rightarrow \mathtt{(tr1}|_{\mathrm{M}}\ \mathtt{=\ tr2}|_{\mathrm{M}}\mathtt{)}$

The conclusion of theorem 6.1 then gives us $\mathtt{P(map(evt;tr}_m\mathtt{))}$ and we must show $\mathtt{P(tr}_u\mathtt{)}$. But this follows from fact that P is preserved by the layer relation and $\mathtt{map(evt;tr}_m\mathtt{)}\ \mathtt{layerR}_{\mathrm{E}}\ \mathtt{tr}_u$, by the full_switch_inv assumption. □

6.1 Switchable Properties

Having proved that out switch preserves all switchable properties, we should show that some interesting properties are switchable. The "smallest" switchable property is $\mathtt{Causal}_{\mathrm{E}} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{\mathrm{E}}$

$\forall\mathtt{E:EventStruct.}\ \mathtt{switchable}_{\mathrm{E}}(\mathtt{Causal}_{\mathrm{E}} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{\mathrm{E}})$

This follows from the fact that both of the properties satisfy all six of the metaproperties in

```
switchable0(E)(P) ==
     safetyR_E preserves P
   ∧ memorylessR_E preserves P
   ∧ (ternary) composableR_E preserves P
   ∧ send-enabledR_E preserves P
   ∧ asyncR_E preserves P
   ∧ delayableR_E preserves P
```

Any property P that satisfies these six meta-proerties can be conjoined with $\mathtt{Causal}_{\mathrm{E}} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{\mathrm{E}}$ to get a switchable property.

$\forall\mathtt{E:EventStruct.}\ \forall\mathtt{P:TraceProperty}_{\mathrm{E}}.$
$\mathtt{switchable0(E)(P)} \Rightarrow \mathtt{switchable}_{\mathrm{E}}(\mathtt{P} \wedge \mathtt{Causal}_{\mathrm{E}} \wedge \mathtt{No\text{-}dup\text{-}deliver}_{\mathrm{E}})$

The following recursively defined relation on lists holds when the two lists agree on the order of the elements they have in common.

```
as ||* bs ==
  case as of
    [] => True
    a::as′ => case bs of
                [] => True
                b::bs′ => ((¬(a ∈ bs)) ∧ as′ ||* bs)
                        ∨ ((¬(b ∈ as)) ∧ as ||* bs′)
                        ∨ ((a = b) ∧ as′ ||* bs′)
             esac
  esac
```

The total-order property is defined by

```
totalorder(E)(tr) ==
  ∀p,q:Label.  map(msg_E;tr delivered at p) ||* map(msg_E;tr delivered at q)
```

This says that the lists of messages delivered to any two locations agree on the order of messages that they have in common. This property is a "local-deliver-property". It only depends on some relation on the lists `tr delivered at p`.

```
local_deliver_property(E;P)(tr) == P(λp.tr delivered at p)
```

We can show that any such property is switchable, provided the relation on the local delivery lists satisfies some closure conditions.

```
∀E:EventStruct. ∀P:Label → (|E| List) → ℙ.
  (∀f,g:Label → (|E| List).  (∀p:Label. g(p) ≤ f(p)) ⇒ P(f) ⇒ P(g))
  ⇒ (∀f,g:Label → (|E| List). (∃a:|E|. ∀p:Label.
        g(p) = filter(λb.(¬(b =msg=_E a));f(p))) ⇒ P(f) ⇒ P(g))
  ⇒ (∀f,g,h:Label → (|E| List).
       (∀p,q:Label.  ∀x∈f(p). ∀y∈g(q). ¬(x =msg=_E y))
       ⇒ (∀p:Label. h(p) = (f(p) @ g(p)))
       ⇒ P(f)
       ⇒ P(g)
       ⇒ P(h))
  ⇒ switchable0(E)(local_deliver_property(E;P))
```

Using this theorem, we check the closure conditions for the relation that defines total order, and all the conditions are met. So we have:

```
∀E:EventStruct. switchable_E(totalorder(E) ∧ Causal_E ∧ No-dup-deliver_E)
```

7 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.
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. 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 networks.
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.

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] M. Aagaard and 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 Verlag, 1993.

[2] 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.

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

[4] 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.

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

[6] J. Hickey, N. Lynch, R. van Renesse. Specifications and proofs for Ensemble layers. In R. Cleaveland, ed., *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 119–133. Springer Verlag, 1999.

[7] 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 Verlag, 1996.

[8] C. Kreitz. Automated fast-track reconfiguration of group communication systems. In R. Cleaveland, ed., *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 104–118. Springer Verlag, 1999.

[9] C. Kreitz, M. Hayden, J. Hickey. A proof environment for the development of group communication systems. In C. & H. Kirchner, eds., *15th Conference on Automated Deduction*, LNAI 1421, pages 317–332. Springer Verlag, 1998.

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

[11] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, R. Constable. Protocol switching: Exploiting meta-properties. In Luis Rodrigues and Michel Raynal, eds., *International Workshop on Applied Reliable Group Communication (WARGC 2001)*. IEEE CS Press, 2001.

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