In Proceedings of the *14th International IFIP Symposium on Protocol Specification, Testing, and Verification*, pages 339–346, Chapmann & Hall, 1994

Compositional Specification and Verification of High-Speed Transfer Protocols

Peter Herrmann, Heiko Krumm

Dept. of Computer Science, Dortmund University, D-44221 Dortmund, Germany

Abstract

Transfer protocols are composed from basic protocol mechanisms and accordingly a complex protocol can be verified by a series of relatively simple mechanism proofs. Our approach applies L. Lamport's Temporal Logic of Actions (TLA). It is based on a modular compositional TLA-style and supports the analysis of flexibly configured high-speed transfer protocols.

Keyword Codes: C.2.2; F.3.1; D.3.3
Keywords: Protocol Verification; TLA; Composition

1 Introduction

Flexible application-driven protocol configuration can help to enhance the performance of high-speed networks [3, 10]. Therefore efficient means for the analysis of protocol spectra are needed. Our approach reflects that the composition of transfer protocols from basic protocol mechanisms corresponds to a structuring of service requirements into different service properties. It applies decomposition and structures the protocol verification into separated and easy-to-understand mechanism proofs.

The approach is based on L. Lamport's Temporal Logic of Actions (TLA [5]) and refers to the concepts of refinement mappings [1] and formal composition by logical conjunction [2]. TLA is well-suited for the needs of practical protocol verification. Protocol designers are familar with state transition based models. The definition and verification of interesting liveness properties is supported by TLA.

Nevertheless, TLA is a very fundamental approach and does not provide for special means which are tailored to the modelling of concurrent process systems. Therefore, we designed a compositional specification style for TLA which is oriented at CCS [7] and Lotos [4]. In comparison with [2], the processes do not interact via shared variables but perform joint actions. This stateless way of interaction has different benefits. Especially resource-oriented processes as well as constraint-oriented processes can be represented (cf. [9]). Furthermore, the style supports decompositional proofs. A system is the logical conjunction of its processes and the style conventions assure the absence of contradictions in the system formula. Therefore process properties directly are inherited to the system.

We applied the compositional style and the decompositional verification method to different transfer protocols. The applications were supported by existing general TLAtools (syntax-directed editor, browser, interpreter, model checker, predicate logic theorem prover frontend) which not yet have been tailored to the style. Yet we made the experience that it is possible to verify complex protocols within few man-month (e.g., the verification of XTP [8] needed 7 weeks, 3 weeks for the design of specifications and proof ideas, 4 weeks for the formal theorem proofs).

At first the paper introduces the style and the verification method. Thereafter some views to the verification of XTP are given in order to examplify the application and to give an impression of the reduced verification complexity. The reader is assumed to be familar with TLA and refinement mappings [5, 1].

2 Compositional specification style

As in CCS and Lotos, a process in principle is an open subsystem but a single process specification can be interpreted for its own. In this case it reflects a closed system consisting of the process and an evironment which is universal in the sense that it does not constrain the process. A process P is defined by a canonical TLA-formula P:

$$\begin{aligned} P \triangleq\ & P.Init \wedge \Box[\exists p \in P.ptype_1 : P.act_1(p) \vee .. \vee \exists p \in P.ptype_n : P.act_n(p)]_{P.V} \\ & \wedge \forall p \in P.ptype_i : WF_{P.V}(P.eact_i(p)) \wedge .. \wedge \forall p \in P.ptype_j : WF_{P.V}(P.eact_j(p)) \\ & \wedge \forall p \in P.ptype_k : SF_{P.V}(P.eact_k(p)) \wedge .. \wedge \forall p \in P.ptype_l : SF_{P.V}(P.eact_l(p)). \end{aligned}$$

The initial predicate $P.Init$ describes the set of starting states. $P.V$ stands for the tuple of private state variables of P. $P.act_i(p : P.ptype_i)$ are the different actions of P which constitute its next-state relation. The actions may be parametrized by data parameters supporting the communication of values between the process and its environment. By style conventions, the actions only affect private variables and must be mutually disjoint in their non-stuttering subrelations.

The liveness properties are described by fairness assumptions on conditioned actions $P.eact_i(p : ptype_i) \triangleq P.act_i(p) \wedge p \in e_i$ where a $P.eact_i$ is the conjunction of the action $P.act_i$ and an environment condition. e_i stands for an additional state variable called environment readiness variable. It is assumed to be set by the environment of P: if $p \in e_i$, the environment can tolerate the action $act_i(p)$ in the next step. Thus the formula $P \wedge \Box(e_1 = P.ptype_1 \wedge .. \wedge e_n = P.ptype_n)$ describes a separated process in an universal environment.

A system S composed of processes $P_1, P_2, .., P_m$ is described by a TLA-formula $S \triangleq P_1 \wedge P_2 \wedge .. \wedge P_m \wedge CC$. The different P_j denote the process formulas. Additionally, there is another conjunctive term, the coupling constraint CC. CC is an invariant and describes the specific coupling of the system. It can be structured into a conjunction of participation constraints P_jC of the different processes: $CC \triangleq \Box(P_1C \wedge .. \wedge P_mC)$. A participation constraint again is a conjunction of two parts: $P_jC \triangleq P_jCON \wedge P_jRED$.

P_jCON constrains the occurrence of P_j-steps in system executions. It is a disjunction of $Unchanged(P_j.V)$ and of action terms $\exists p \in P_j.ptype_i : (P_j.act_i(p) \wedge PeerActions) \wedge StutteringRest$ which are introduced for each action $P_j.act_i$ of P_j.

$PeerActions$ is a conjunction of actions of other processes which shall contribute to the same joint action: $PeerActions \triangleq P_k.act_o(p) \wedge .. \wedge P_l.act_q(p)$. If $P_j.act_i$ is an internal action, i.e., if it is not involved in joint actions, then $PeerActions$ equals to $true$.

$StutteringRest$ is a conjunction of $Unchanged$-statements for processes $P_r, .., P_s$ which are not involved in a joint action with $P_j.act_i$. It describes the interleaving atomicity of $P_j.act_i$ and may be set to $true$ if parallelism shall be tolerated with respect to logically non-connected actions as well. Furthermore, it is possible to postulate the interleaving atomicity of $P_j.act_i$ only with respect to some subset of the other processes. In order to keep the system formula simple, we recommend to introduce interleaving as strict as it is possible with respect to a specific system of interest.

The other part of P_jC, P_jRED states the substitution of the environment readiness variables $P_j.e_i$. It has to be chosen in accordance with the joint action terms of P_jCON and is a conjunction of equations. For each interface action $P_j.act_i$ of P_j, an equation $P_j.e_i = \{p : Enabled(P_k.act_o(p)) \wedge .. \wedge Enabled(P_l.act_q(p))\}$ has to be introduced where the processes and actions referenced are those of $PeerActions$ of the corresponding action term. For internal actions $P_j.act_i$ the equation $P_j.e_i = P_j.type_i$ is introduced.

By style convention, we claim that the different fairness assumptions of the process actions, contributing to the same joint action, fit together, i.e., all process actions of the same joint action must either be weak fair, strong fair, or without any fairness condition.

The compositional system formula $S \triangleq P_1 \wedge P_2 \wedge .. \wedge P_m \wedge CC$ can be transformed syntactically into an equivalent 'flat' canonical formula

$$\begin{aligned} S \triangleq\ & S.Init \wedge \Box[\exists p \in ptype_1 : S.act_1(p) \vee .. \vee \exists p \in ptype_n : S.act_n(p)]_{S.V} \\ & \wedge \forall p \in ptype_i : WF_{S.V}(S.eact_i(p)) \wedge .. \wedge \forall p \in ptype_j : WF_{S.V}(S.eact_j(p)) \\ & \wedge \forall p \in ptype_k : SF_{S.V}(S.eact_k(p)) \wedge .. \wedge \forall p \in ptype_l : SF_{S.V}(S.eact_l(p)) \end{aligned}$$

$S.Init$ is the conjunction of the processes' $Init$-predicates. The actions $S.act_i$ are conjunctions of process actions and $Unchanged$-statements (guided by the P_jCON). Due to the style conventions the fairness assumptions of system actions are inherited from the process actions' fairness.

The specification style is defined in terms of TLA+ [6] and takes profit from TLA+ modules. A tool can support the definition of system structures (e.g., interactive graphical editing of coupling constraints) and can compute the flat formula.

3 Structured verification

To prove that a protocol P implies a service S, decompositions PC and SC can be used. PC is composed of a set of protocol mechanisms PP_j: $PC \triangleq PP_1 \wedge PP_2 \wedge .. \wedge PP_n \wedge PCC$. Correspondingly, SC is composed of a set of service properties SP_j: $SC \triangleq SP_1 \wedge SP_2 \wedge .. \wedge SP_n \wedge SCC$. Each single protocol mechanism PP_i provides for one functional service property SP_i.

The protocol verification has to prove the implications $P \Rightarrow PC$, $PC \Rightarrow SC$, and $SC \Rightarrow S$. The proof of the decomposition correctness ($P \Rightarrow PC$ and $SC \Rightarrow S$) is easy since both sides of the implications are strongly related. Moreover, $P \Rightarrow PC$ can be splitted (PC is a conjunction). The difficult task of the verification is the proof of $PC \Rightarrow SC$. Due to the corresponding decompositions, this proof can be structured into n mechanism proofs $PP_i \Rightarrow SP_i$ and one additional coupling proof $PCC \Rightarrow SCC$. The coupling proof is a pure safety proof and can be performed quite mechanically because one can profit from those intermediate results of mechanism proofs which describe the action-structure of the refinement mapping. The mechanism proofs are much simpler than a monolithical proof because only a subset of variables and actions has to be regarded. Furthermore, the multitude of present transfer protocols is faced by a relatively small number of basic mechanisms. Therefore mechanism proofs can be re-used respectively can be replaced by references to former proofs.

4 Example

Some aspects of the data transfer of XTP [8] shall exemplify the approach. As it is outlined in Fig. 1, the transfer service S can be specified by a FIFO message queue per transfer direction. The action *submit* models the request of the transfer of a message i from site s to site d which is represented by an enqueuing operation. The parameter *nodc* denotes the 'no data corruption control' flag. The dequeuing action *deliver* models the indication at site *dest*.

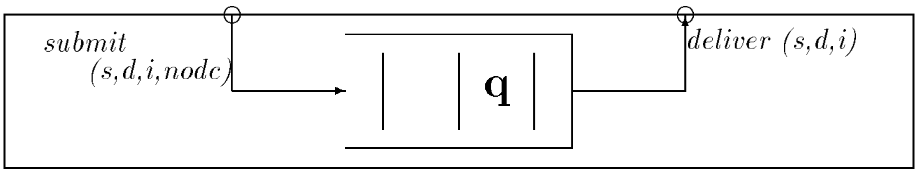

Figure 1: Monolithical Service

The compositional service specification SC is composed of service constraint processes *No Corruption*, *No Gaps*, and *No Duplicates*. To simplify the verification, we have chosen already refined models of service constraints. They reflect a protocol-near distribution of variables over the three parties *transmitter*, *medium*, and *receiver*.

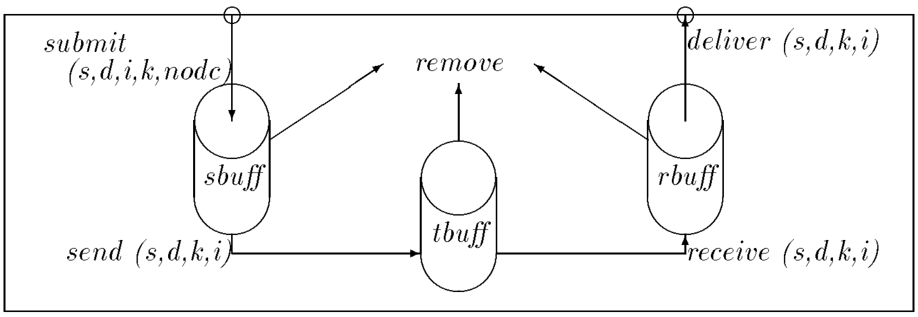

Figure 2: Service Constraint *No Corruption*

Fig. 2 outlines the service constraint process *No Corruption*. It tolerates various transfer errors but does only permit corruptions if the parameter *nodc* is set. The set-type state variables *sbuff*, *tbuff*, and *rbuff* represent message buffers of the three parties. Messages are represented by tuples of transmitter address s, receiver address d, message identification key k, and user data i. As in Fig. 1, the actions *submit* and *deliver* model transfer requests and indications now by set insertion and member selection operations to tolerate reordering and duplication. The other actions are internal. They model loss and the forwarding of buffer elements.

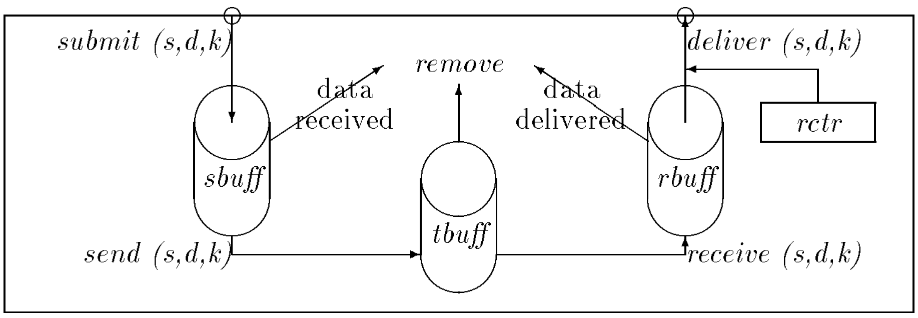

Figure 3: Service Constraint *No Gaps*

In Fig. 3, the second service constraint process *No Gaps* is outlined. At first it models the safety aspect of the absence of loss, namely that the sequence of packets delivered is free from gaps. Indications are controlled by the receive counter *rctr*. Secondly the liveness aspect is reflected by exclusion of the loss of non-delivered packets (*remove* restrictions) in connection with fairness assumptions.

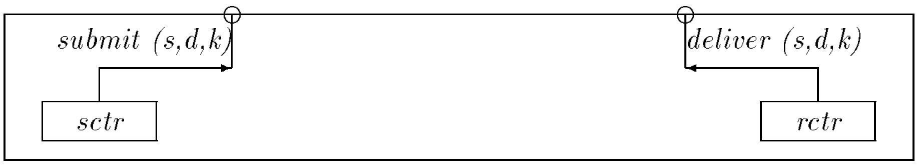

Figure 4: Service Constraint *No Duplicates*

The third service constraint process *No Duplicates* is outlined in Fig. 4. The send counter *sctr* assigns keys *k* to the messages so that the receiver can detect duplications.

The compositional service specification *SC* can be built now by a composition of the three constraint processes, the coupling of which defines the equally-named actions of the processes to be performed jointly. *SC* implies the monolithic queue model. The main task of the proof is the design of an invariant which relates the buffer contents and counter variables. The idea behind this invariant guides the design of the refinement mapping as well.

After describing XTP by a monolithical protocol specification P we design a compositional protocol system *PC*. *PC* is a composition of three subsystems PP_{NC}, PP_{NG}, and PP_{ND} which correspond with the three service constraints *No Corruption*, *No Gaps*, and *No Duplicates*. Two of the PP_x again are compositions, namely of a model of the underlying network *Wire* and models of the protocol mechanisms *Buffers*, *Selective Repeat*, and *Keep Order*. PP_{NC} consists of *Wire* and *Buffers*; PP_{NG} of *Wire*, *Buffers*, and *Selective Repeat*; and PP_{ND} of *Keep Order*. To simplify this example, *Wire* is reliable besides of loss and the protocol mechanisms are not structured further.

The proof of $P \Rightarrow PC$ is straightforward and consists of the proof of the four implications $P \Rightarrow PP_x$ and $P \Rightarrow PCC$ (since the PP_x are compositions the proof of each $P \Rightarrow PP_x$ can be splitted again).

To give some impression on the essential tasks of the verification, we now look at basic components of the protocol system. So *Wire* is described as follows (canonical process formula omitted).

```
Wire
type   Adr, Packet
var    inq : array [Adr] of queue of Packet
----------------------------------------------------------------
init   $\forall$ x $\in$ Adr : inq[x] = empty
action send(p : Packet, src, dest : Adr) $\triangleq$
          $\wedge$ inq[dest]$'$ = enqueue(inq[dest],p)
          $\wedge$ p.src = src $\wedge$ p.dest = dest
action receive(p : Packet, src, dest : Adr) $\triangleq$
          $\wedge$ p = firstqueue(inq[dest])
          $\wedge$ inq[dest]$'$ = dequeue(inq[dest])
          $\wedge$ p.src = src $\wedge$ p.dest = dest
action loss $\triangleq$ $\exists$ x $\in$ Adr : inq[x]$'$ = dequeue(inq[x])
```

The network model consists of an array of message queues *inq[dest]*, keeping the *dest*-directed XTP-PDUs. The actions *send* and *receive* model the transfer request and transfer indication of PDUs. The loss of PDUs is expressed by the action *loss*, removing PDUs non-deterministically from the queues.

```
Buffers
parm   Station : Adr
type   Adr, Info, Packet, Key
var    SB : array [Adr,Key] of [[i : Info $\cup$ {$\perp$}]]
       RB : array [Adr,Key] of [[i : Info $\cup$ {$\perp$}]]
----------------------------------------------------------------
init   $\forall$ x $\in$ Key, a $\in$ Adr: SB[a,x].i = $\perp$ $\wedge$ RB[a,x].i = $\perp$
action submit(i : Info, dest : Adr, k : Key) $\triangleq$ SB[dest,k].i = $\perp$ $\wedge$ SB[dest,k].i$'$ = i
action deliver(i : Info, src : Adr, k : Key) $\triangleq$ RB[src,k].i = i
action send_info(p : Packet, dest : Adr, k : Key) $\triangleq$
          $\wedge$ SB[dest,k].i $\neq$ $\perp$
          $\wedge$ p.src = Station $\wedge$ p.dest = dest $\wedge$ p.he.key = k
          $\wedge$ p.type = info $\wedge$ p.info = SB[dest,k].i
action send_ctrl(p : Packet, dest : Adr) $\triangleq$
          p.src = Station $\wedge$ p.dest = dest $\wedge$ p.type = ctrl
action rec_info(p : Packet, src : Adr, k : Key) $\triangleq$
          $\wedge$ RB[src,k].i = $\perp$
          $\wedge$ RB[src,k].i$'$ = p.info
          $\wedge$ p.src = src $\wedge$ p.dest = Station $\wedge$ p.type = info $\wedge$ p.he.key = k
action rec_ctrl(p : Packet, src : Adr) $\triangleq$
          p.src = src $\wedge$ p.dest = Station $\wedge$ p.type = ctrl
action sendremove(dest : Adr, k : Key) $\triangleq$ SB[dest,k].i $\neq$ $\perp$ $\wedge$ SB[dest,k].i$'$ = $\perp$
action rcvremove(src : Adr, k : Key) $\triangleq$ RB[src,k].i $\neq$ $\perp$ $\wedge$ RB[src,k].i$'$ = $\perp$
```

The process *Buffers* describes one basic XTP protocol entity and will be parametrized by the address *Station* of the site assigned to. It contains a send buffer *SB* and a receive buffer *RB* including messages which are indexed by the address of the station, the address of the transfer partner, and the message key. $\perp$ denotes empty buffer elements. The communication with the user of the protocol is described by the actions *submit* and *deliver*. Because XTP distinguishes between control and information packets, sending and receiving is split into actions *send_info*, *send_ctrl*, *rec_info*, and *rec_ctrl*. The removal of buffer elements is described by the actions *sendremove* and *rcvremove*.

$$\begin{array}{l}
\text{RM} \triangleq \ \wedge\ \text{sbuff} = \{\, [[\text{src,dest,k,i}]]\ :\ \text{S[src].SB[dest,k]} = \text{i}\} \\
\qquad\quad\ \wedge\ \text{tbuff} = \{\, [[\text{src,dest,k,i}]]\ : \\
\qquad\qquad\qquad\qquad \exists\ \text{p} : \ \wedge\ \text{inqueue(inq[dest],p)}\ \wedge\ \text{p.src} = \text{src} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \text{p.he.key} = \text{k}\ \wedge\ \text{p.info} = \text{i}\ \wedge\ \text{p.type} = \text{info}\} \\
\qquad\quad\ \wedge\ \text{rbuff} = \{\, [[\text{src,dest,k,i}]]\ :\ \text{S[dest].RB[src,k]} = \text{i}\}
\end{array}$$

The composition of the mechanisms *Wire* and *Buffers* builds PP_{NC} and can be proved to imply the service constraint *No Corruption* by means of the refinement mapping above. The variables of a station *st* are identified by the qualifier *S[st]*.

The mechanism *Selective Repeat* detects lost data and performs selective-repeat retransmission. The specification is not shown in detail here. It provides for the saving of copies of messages sent, a receive counter, *span*-list acknowledgements, notification, and time-out based loss-detection.

$$\begin{array}{l}
\text{action S[src].send\_info(p : Packet, dest : Adr, k : Key, rseq : BOOL)} \triangleq \\
\qquad\qquad \wedge\ \text{Wire.send(p,src,dest)} \\
\qquad\qquad \wedge\ \text{Buffers(src).send\_info(p,dest,k)} \\
\qquad\qquad \wedge\ \text{SelectiveRepeat(src).send\_info(p,dest,k,rseq)} \\
\text{action S[src].send\_info(p : Packet, dest : Adr, k : Key, rseq : BOOL)} \triangleq \\
\qquad\qquad \wedge\ \text{S[src].SB[dest,k].i} \neq \bot \\
\qquad\qquad \wedge\ \text{inq[dest]}' = \text{enqueue(inq[dest],p)} \\
\qquad\qquad \wedge\ (\text{timeout(S[src].t[dest])} \Rightarrow \text{rseq}) \\
\qquad\qquad \wedge\ \text{S[src].t[dest]}' = \text{IF rseq THEN start ELSE S[src].t[dest]} \\
\qquad\qquad \wedge\ \text{p.src} = \text{src}\ \wedge\ \text{p.dest} = \text{dest}\ \wedge\ \text{p.he.key} = \text{k}\ \wedge\ \text{p.type} = \text{info} \\
\qquad\qquad \wedge\ \text{p.info} = \text{S[src].SB[dest,k].i}\ \wedge\ \text{p.tr.f.rseq} = \text{rseq}
\end{array}$$

PP_{NG} is a composition of *Wire*, *Buffers*, and *Selective Repeat* and implies the service property *No Gaps*. The specification above reflects a clipping of the flat form of PP_{NG}. It shows the system action *S[src].send_info* as joint action of process actions and its expansion. The process action *Buffers(src).send_info* equals to the action *send_info* of process *Buffers* under substitution of parameter *Station* by *src*.

$$\begin{array}{l}
\text{Inv} \triangleq \forall\ \text{p,src,dest,k} : \\
\qquad\qquad \vee\ \text{S[src].SB[dest,k]} \neq \bot\ \wedge\ \neg\ \text{S[src].SB.Keep[dest,k]} \\
\qquad\qquad \vee\ \wedge\ \text{inqueue(inq[dest],p)}\ \wedge\ \text{p.src} = \text{src}\ \wedge\ \text{p.dest} = \text{dest} \\
\qquad\qquad\quad\ \wedge\ \text{p.type} = \text{ctrl}\ \wedge\ \text{p.tr.f.rseqresp}\ \wedge\ \text{k} \in \text{p.c} \\
\qquad\qquad \vee\ (\text{src,k}) \in \text{S[dest].Rcv} \\
\qquad\qquad\quad \Rightarrow \text{S[dest].RB[src,k].i} \neq \bot\ \vee\ \text{S[dest].Rctrg[src]} > \text{k} \\
\text{RM} \triangleq \ \wedge\ \forall\ \text{src,dest} : \text{rctrg[src,dest]} = \text{S[dest].Rctrg[src]} \\
\qquad\quad\ \wedge\ \text{sbuff} = \{\, [[\text{src,dest,k}]]\ :\ \text{S[src].SB[dest,k].i} \neq \bot\} \\
\qquad\quad\ \wedge\ \text{tbuff} = \{\, [[\text{src,dest,k}]]\ : \\
\qquad\qquad\qquad\qquad \exists\ \text{p} : \ \wedge\ \text{inqueue(inq[dest],p)}\ \wedge\ \text{p.src} = \text{src} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \text{p.dest} = \text{dest}\ \wedge\ \text{p.type} = \text{info}\ \wedge\ \text{p.he.key} = \text{k}\} \\
\qquad\quad\ \wedge\ \text{rbuff} = \{\, [[\text{src,dest,k}]]\ :\ \text{S[dest].RB[src,k].i} \neq \bot\}
\end{array}$$

For the proof of $PP_{NG} \Rightarrow NoGaps$ we need the invariant *Inv* and the refinement mapping *RM* as defined above. *Inv* assures that a message can only be confirmed to the transmitter and removed from its send buffer if it is correctly received. The refinement mapping is merely equal to that of the proof of *No Corruption*.

The protocol mechanism *Keep Order* simply contains two counters: *Sctr* assigns unambiguous keys to submitted messages in incremental order, *Rctr* prevents the delivery of reordered or duplicated messages. Due to the close relation to the service constraint, the proof of $PP_{ND} \Rightarrow NoDuplication$ is very simple.

Finally, the coupling proof has to be performed to complete the verification. It can be performed quite mechanically and is not outlined here.

5 Conclusion

We introduced a practicable approach for the compositional verification of transfer protocols which provides a framework for the investigation of flexible protocol configurations. Present work concentrates on the establishment of libraries of basic protocol mechanisms PP_i, service properties SP_j, and valid implications $PP_i \Rightarrow SP_j$. Therefore, the verification of a specific protocol can be performed by the design of equivalent compositional protocol and service specifications to which only the coupling proof has to be added.

A problem remains: often already a composition of protocol mechanisms is needed to provide even a single service property (e.g., *No Gaps* in the XTP-example). Therefore, a library of mechanism combinations would be of interest. Instead of introducing this larger library of mechanism combinations, we are solving the problem on the service side. We represent abstract service properties (e.g., *No Gaps*) by a composition of more basic service constraints so that these service constraints correspond one-to-one with single basic protocol mechanisms.

References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[3] Z. Haas. A protocol structure for high-speed communication over broadband ISDN. *IEEE Network Magazine*, pages 64–70, Jan. 1991.

[4] ISO. *LOTOS: Language for the temporal ordering specification of observational behaviour*, International Standard ISO/IS 8807 edition, 1987.

[5] L. Lamport. The Temporal Logic of Actions. Technical Report 79, DEC Digital Systems Research Center, Palo Alto, May 1991. Research Report. To appear in *ACM TOPLAS*.

[6] L. Lamport. TLA+: Syntax and Semantics. To appear, Preliminary Version, DEC Digital Systems Research Center, Palo Alto, Feb. 1992. Research Report.

[7] R. Milner. *A Calculus for Communicating Systems.* Lecture Notes in Computer Science 92. Springer, Berlin, 1980.

[8] Protocol Engines, Incorporated. *XTP Protocol Definition Revision 3.4*, 1989.

[9] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Agarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification*, volume VIII, pages 189–204, Elsevier, 1988. IFIP.

[10] M. Zitterbart, B. Stiller, and A. N. Tantawy. Application-driven flexible protocol configuration. In N. Gerner, H.-G. Hegering, and J. Swoboda, editors, *Kommunikation in Verteilten Systemen*, pages 144–158, 8. Fachtagung, München, Mar. 1993. GI/ITG, Springer-Verlag.