In Proceedings of the *15th International IFIP Symposium on Protocol Specification, Testing, and Verification*, pages 171–186, Chapmann & Hall, 1995

Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations

P. Herrmann and H. Krumm
University of Dortmund, Dept. of Computer Science
D-44221 Dortmund, Germany
Phone: +49-231-7554674, Fax: +49-231-7554730
Internet: {herrmann|krumm}@ls4.informatik.tu-dortmund.de

Abstract

Presently, many communication protocols are under development which are tailored to the efficient high-speed data transfer meeting different application-specific requirements. Our approach concentrates on a framework which facilitates the formal verification of the protocols. The framework supplies verified and re-usable implications between predefined protocol and service specification components. For the verification of a specific protocol, protocol, service and medium can be modelled as compositions of framework specification components. The verification corresponds to proving that the system of protocol and medium implies the service. This implication can be proven by combining component implications of the framework. We apply L. Lamport's Temporal Logic of Actions (TLA) and use a TLA specification style supporting the compositional specification of process systems and the inference of system properties from process properties.

Keywords

Composition, examples, framework, protocol synthesis, TLA, verification

1 INTRODUCTION

Modern high-speed transfer protocols are composed from basic protocol mechanisms which correspond to service requirements such that a certain service property is provided by a corresponding subset of the protocol mechanisms. The protocol designer considers the requirements of the service to be provided, the assumptions of the medium used, and aspects of efficient implementation. He selects a suitable set of protocol mechanisms and designs a combination of them which forms the protocol (eg., Clark and Tennenhouse (1990), Doeringer et al. (1990), Haas (1991), O'Malley and Peterson (1991)). Moreover, recent approaches propose the application-driven flexible configuration of high-speed transfer protocols (La Porta and Schwartz (1993), Plagemann et al. (1993), Zitterbart et al. (1993)). The protocol is configured from a set of modules at runtime. During design, a spectrum of possible service requirements has to be considered. A set of protocol mechanism modules has to be designed as well as rules for the requirement-dependent module selection and the consistent configuration of modules. Here again, each member of the designed set of protocol variants is composed of a set of basic protocol mechanisms and an identifyable correspondence between properties of the service provided and protocol mechanism subsets exists.

Our approach is tailored to the verification of such protocols. Even in different protocol development projects, the same spectra of functional service requirements, protocol mechanisms, and medium properties are of interest. This leads to the idea of supporting formal protocol specification and verification by a framework of re-usable predefined components. The framework supplies specification as well as verification elements. The specification elements are modular definitions of process types. Service and protocol descriptions are process compositions consisting of parametrized instances of these process types. The verification elements are implications between process systems and processes and can be used as theorems for protocol verification.

While the sets of service property processes and protocol mechanism processes are relatively small, there is a large space of possible configurations. In order to reflect the broad variety of interesting protocols and services, process instances can result from specialisations of process types and they can be coupled flexibly to form process systems. With respect to verification, there mostly is no one-to-one correspondence between protocol mechanisms and service properties. Special compositions of several protocol mechanisms are needed to provide certain service properties. So, direct implications do not exist which could be used as basic and re-usable verification elements. Therefore we are looking for re-usable subimplications. We structure the 'gap' between protocol and service and propose a layered framework:

- SCs (Service Constraints): SC processes abstractly model properties of the service provided, e.g., no duplication, no corruption, live transfer. The processes describe these properties directly by means of global non-distributed models.
- APMs and AMCs (Abstract Protocol Mechanisms and Abstract Medium Constraints): APMs introduce the distribution of functionality in principle. Functions are located to sites and coupled via the exchange of PDUs (protocol data units). The medium, the PDU-communication, and the local control structures are very abstract images of elements of real protocols. APMs define isolated views of these elements such that each view is devoted to an identifiable contribution to the provision of a certain SC. The intertwining of mechanisms which we find in real protocols is resolved by use of idealised control structures (e.g., infinite domain of sequence numbers, multiple buffering of the same PDU, infinite memory of processes). APM-theorems provide valid implications

between APM-subsystems and single SCs.

- FAPMs and FAMCs (Finite Abstract Protocol Mechanisms and Finite Abstract Medium Constraints): FAPMs build the bridge between PMs and APMs. They provide views which on the one hand are based on finite control structures and thus are near images of real protocol mechanism configurations. On the other hand, there is a direct contribution of FAPM subsets to single APMs. FAPM-theorems provide valid implications between FAPM-subsystems and single APMs.
- PMs and MCs (Protocol Mechanisms and Medium Constraints): They model single protocol mechanisms and properties of the underlying medium which are components of real protocols, e.g., frame check sequence provision and filtering of PDUs, sequence number provision, receiver buffering and resequencing, live acceptance of PDUs, finite capacity of the medium, live transfers in the medium. PM-theorems provide valid implications between PM-subsystems and single FAPMs.

The verification of a given protocol against given service requirements is performed in three steps, service decomposition, protocol decomposition, and compositional proof. Service decomposition results in a service model which is a composition of SC processes and implies the conjunction of the given service requirements. Protocol decomposition results in a system of PM and MC processes which is implied by the given protocol. The decompositions are correct if the two corresponding implications can be proven. The proofs of the two implications are relatively simple since PMs are closely related to real protocol mechanisms and SCs are closely related to service requirements. Thus, the dominating task of protocol verification is the compositional proof, i.e., one has to prove that the protocol composition implies the service composition. This task is facilitated by the theorems of the framework. The theorems supply valid subimplications which transitively contribute to the proof. Therefore, the proof can be accomplished by the instantiation and arrangement of framework verification elements.

The approach of supporting design and verification by an application field specific theory is well-known but hard to follow for the broader field of data transfer protocols, since there are complex relations between protocol mechanisms and service requirements which depend on details of the configuration of an interesting protocol. Therefore, there is a trade-off between the required generality of re-usable elements and the wide spectrum of specialisations used in real protocols. Modelling a protocol by means of a small set of re-usable specification modules tends to introduce complex module instantiations and inter-modul relations. Theorems of general validity tend to be too weak to support the verification of special protocols.

We propose a solution of these problems which is based on a suitable combination of several approaches. On the technical side, we apply a modular specification technique which supports flexible process composition. The technique also is compositional with respect to verification. Theorems of subsystems remain valid if the subsystems are embedded in systems. On the side of the application field 'transfer protocols', we propose a suitable structuring by means of the framework layers and components.

In particular, we address the important problem of modular liveness proofs. While safety proofs can be performed easily in a modular way, since safety properties of subsystems are inherited to any embedding systems, liveness properties of subsystems can be disturbed by environments. Therefore, compositional liveness theorems of subsystems contain conditions on the environment and the theorems are of use for the proof of the system's liveness only if these conditions can be proven to be valid in the system. The proof of the conditions may not be modular since it considers the whole system. To solve this problem, we only use liveness theorems, the environment conditions of which are safety properties of the system. Moreover, already during the design of a liveness theorem, we consider each process type of the corresponding layer of the framework as possible component of the environment and only propose liveness theorems, the environment conditions modularily can be proven in each interesting environment. Actually, the consideration of the possible co-processes already takes place during the design of those process types of the framework which model liveness properties, since liveness theorems are implications between these process types. So, liveness process types are designed to model liveness assumptions as weak as possible.

The work is based on L. Lamport's Temporal Logic of Actions (TLA) (Lamport, 1994), its state transition based modelling of systems, its verification means, and its handling of liveness properties by fairness assumptions on the scheduling of actions. Particularly, we use an enhancement of the compositional TLA specification style introduced in Herrmann and Krumm (1994), Mester and Krumm (1995). Systems are composed from processes which interact via joint actions similarly to LOTOS processes (ISO, 1987) while Abadi and Lamport (1993) address the use of shared state variables. Syntactical style conventions ensure the compositionality of process properties supporting the modular structuring of proofs (Herrmann and Krumm, 1994).

The layered framework components reflect a stepwise refinement of abstract global system views to implementation-near distributed views which is pursued by many approaches for the formal construction of general distributed systems (e.g., Kurki-Suonio and Järvinen (1988)). Particularly, Shankar and Lam (1992) present heuristics for the stepwise construction of distributed systems and report on the corresponding design of correct sliding window protocols (Moreover, the modelling and verification technique behind is not far away from TLA). While the construction approaches focus on the relatively ambitious objective of systematically integrating design and verification steps, our work yet concentrates on the 'a posteriori' verification of given protocol designs which is also supported by reachability analysis based protocol verification approaches (e.g., Holzmann (1990)). In comparison with present general construction approaches, our approach renders a higher degree of user support. Due to specialisation, e.g., on transfer protocols, one can supply re-usable specification and verification elements. In comparison with automated reachability analysis tools, verifications are not restricted to models with manageable finite state space and liveness properties can be modelled more flexibly since symbolic temporal logic reasoning is applied.

The over-all approach includes work of several types. This paper outlines and concentrates on the re-usable verification elements which are implications between systems of pre-defined process types. At first, the paper gives short introductions into the compositional TLA-style applied and the corresponding structured verification technique. Thereafter, the structure of the framework is outlined and exemplified. Finally, we describe the concepts of re-usable theorems in more detail and give an outline of their application. The examples used in the paper are chosen from the process type and theorem library of the transfer protocol framework. Presently, the library reflects the transfer-phase of two-party protocols and supports the verification of sliding-window protocol variants. Extensions of the library are under development which model connection management and multi-party interactions.

2 TLA AND COMPOSITIONAL SPECIFICATION STYLE

TLA (Lamport, 1994) is a linear time temporal logic modelling systems as state transition systems. The state space is defined by a set of variables v. A canonical TLA formula is a conjunction of three parts: $Init \wedge \Box[Next]_v \wedge Fair$. $Init$ is a condition on the variables v and defines the set of starting states. $[Next]_v$ stands for $(Next \vee v = v')$. $Next$ is a disjunction of so-called actions, where an action is a condition on the variables v and 'primed' variables v' ($\Box$ denotes the 'always' operator). The primed variables refer to the successor state. The first two parts of the formula describe the safety properties of the system. Each trajectory $\langle s_0, s_1, s_2, \ldots\rangle$ starts with s_0 in $Init$ and always two successive states s_i, s_{i+1} are identical (i.e., the system performs a stuttering step) or fulfill $Next$. The third part $Fair$ is a conjunction of fairness assumptions on actions. $WF(\alpha)$ states that the action α is assumed to be scheduled weak fair in a trajectory (i.e., if α is not disabled infinitely often, it occurs infinitely often). $SF(\alpha)$ states that it is scheduled strong fair (i.e., if α is enabled infinitely often it occurs infinitely often). An action α is enabled in a state s (denoted by $Enabled(\alpha)$), if a state s' exists fulfilling the condition $\alpha(s, s')$. The fairness assumptions are a means of expressing liveness properties without implicitly giving additional safety constraints.

Verification corresponds in TLA to the proof of implications. If the formula $(\exists x : F) \Rightarrow (\exists y : G)$ can be proven, the system F (defined on variables x and z) implements the system G (defined on variables y and z), i.e., each projection of a trajectory of F to the visible variables z is a member of the set of the projections of the trajectories of G to z (the operator $\exists$ denotes the hiding of internal variables). Proofs are supported by the TLA inference rules. Moreover, Refinement Mappings (Abadi and Lamport, 1991) can be used. If a refinement mapping $f : (z, x) \rightarrow (z, y)$ exists, the implication above is valid. The mapping f is a refinement mapping with respect to safety, if it maps initial states of F to initial states of G, actions of F to actions or stuttering steps of G, and visible state components z by identity. With resprect to liveness, each image of a trajectory of F has to fulfill the fairness assumptions of G.

We apply a compositional TLA specification style ((Herrmann and Krumm, 1994) with modified style conventions (Mester and Krumm, 1995)) supporting the definition of parametrized process and system types. Systems are composed from processes which interact via joint actions like LOTOS processes (ISO, 1987). As in LOTOS, processes can model implementation parts as well as logical system constraints (cf. Vissers et al. (1988)). The syntax of our examples is a combination of Modula-2 and TLA+ (Lamport, 1992). Fig. 1 gives three examples of process type definitions (before, the data type *key* is defined as a set of natural numbers with the special element <*notsent*>). The header of a process type definition declares the type name and optional parameters. The parameters must be invariant, i.e., of sort CONST in terms of TLA+. Thereafter, the private variables are declared, followed by the *Init* condition. The action definitions are introduced by the keyword *actions*. Additionally, if there are fairness assumptions, a process type definition may supply lists of action names like the third process type *LiveIn*. Only process types contain fairness statements, which represent liveness constraints. By style conventions,

```
type key ≜ nat union {<notsent>} ;

process SDUId ;
  variables cRq : key;
  Init ≜ cRq=0 ;
  actions
    Rq (krq : key) ≜ krq=cRq ∧ cRq′=cRq+1 ;
    In (krq : key) ≜ (krq=<notsent> ∨ krq<cRq) ∧ cRq′=cRq;
  end;

process Corruptions (usd : datatype; tc : reflexive relation on (usd,usd));
  variables buf : set of [[key, usd]] ;
  Init ≜ buf=∅;
  actions
    Rq (krq : key; d : usd) ≜ buf′=buf union { [[krq,d]] } ;
    In (krq : key; d : usd) ≜
     (krq=<notsent> ∨ forall e ( [[krq,e]] notin buf) ∨ exists e ( [[krq,e]] in buf ∧ tc(e,d))) ∧ buf′=buf ;
  end;

process LiveIn (usd : datatype ; ts : nat ; c : nat union {∞});
  variables cRq, maxIn : key;
  Init ≜ cRq=0 ∧ maxIn=0 ;
  actions
    fRq (krq : key) ≜
      (cRq<maxIn+c ∨ c=∞) ∧ krq=cRq ∧ cRq′=cRq+1 ∧ maxIn′=maxIn;
    nRq (krq : key) ≜
      ((cRq >= maxIn+c ∧ c≠∞) ∨ krq≠cRq) ∧ cRq′=cRq+1 ∧ maxIn′=maxIn;
    fIn (krq : key; d : usd ) ≜
      krq=maxIn ∧ maxIn<cRq-min(ts, c-1) ∧ maxIn′=maxIn+1 ∧ cRq′=cRq ;
    nIn (krq : key) ≜
      (krq≠maxIn ∨ maxIn >= cRq-min(ts, c-1))
       ∧ maxIn′=if (krq=<notsent>) then maxIn else max(maxIn, krq+1) ∧ cRq′=cRq;
  WF: fIn;
  end;
```

Figure 1 Example process type definitions.

the actions of a process can only reference private variables of the process and fair actions have to be disjoint to stuttering steps and other actions of the process.

The first process type of Fig. 1 is an auxiliary safety constraint which later on will be used to introduce unique keys for service interface data units (SDUs). An instance *S: SDUId* represents a process, which maintains one private state variable $S.cRq$ and can perform actions $S.Rq$ and $S.In$. $S.Rq$ models data transfer request events, $S.In$ data transfer indication events of the interface of a transfer service. The variable $S.cRq$ counts the occurrences of $S.Rq$ actions. By means of the parameter krq of $S.Rq$, unique identifiers (UIs) $0, 1, \ldots$ are assigned to the occurrences of $S.Rq$, i.e., only action sequences $\langle S.Rq(0), S.Rq(1), S.Rq(2), \ldots \rangle$ can occur. The action $S.In$ may deliver phantoms ($krq = < notsent >$) or SDUs sent before by $S.Rq$. A process of the second type, e.g., $C : Corruptions(usd, identity)$, again models a safety constraint of an unidirectional two-

```
process  Scomp (usd : datatype ; tc : reflexive relation on (usd,usd) ; ts : nat ) ;
  processes S : SDUId ; C : Corruptions(usd, tc) ; L : LiveIn(usd, ts, ∞) ;
  actions
    fRq (krq : key; d : usd) ≙ S.Rq(krq)  ∧  C.Rq(krq,d)  ∧  L.fRq(krq) ;
    nRq (krq : key; d : usd) ≙ S.Rq(krq)  ∧  C.Rq(krq,d)  ∧  L.nRq(krq) ;
    fIn (krq : key; d : usd) ≙ S.In(krq)  ∧  C.In(krq,d)  ∧  L.fIn(krq,d) ;
    nIn (krq : key; d : usd) ≙ S.In(krq)  ∧  C.In(krq,d)  ∧  L.nIn(krq) ;
  end ;
```

Figure 2 Example process composition.

party transfer service. It constrains the delivery of SDUs (action $C.In$). Only SDUs can be delivered which are phantoms or which are identical to the SDUs sent. Weaker corruption constraints, e.g., similarity, can be expressed by other settings of the parameter tc. A process of the third type, e.g., $L : LiveIn(n, 0, \infty)$ models a liveness constraint. Again there are actions Rq and In. Each action is present by two disjunct subactions. The action fIn is supplied with a fairness assumption (i.e., *WF: fIn*), the action nIn not. The fairness assumption of fIn constrains the service to perform SDU deliveries lively. The service stimulating action Rq is splitted also in two subactions fRq and nRq. None of them are supplied with fairness assumptions. The splitting prepares liveness assumptions of the sending user party. The parameters ts and c of $LiveIn$ refer to parameters of other possible safety constraints (ts: deliveries have to be performed only if the maximum number of SDUs being sent but not received exceeds ts; c: buffering capacity of the service).

The translation of safety processes to TLA formulas is straightforward, e.g., C is described by the TLA formula $C.Init \wedge \Box[\exists k, d : C.Rq(k, d) \vee \exists k, d : C.In(k, d)]_{C.buf}$. The translation of liveness processes corresponds to this translation. Additionally, the conjunct *Fair* has to be appended. *Fair* is not simply a conjunction of process action fairness assumptions since the environment of a process may block process actions. To model this, for each fair action, an additional state variable, the so-called environment readiness variable is introduced, e.g., $L.e_{fIn}$ *: set of* $[[key, usd]]$. The variable is shared between the process and its environment. It is written by the environment and read by the process. It is assumed to contain always the action parameter values, the environment of the process is ready to tolerate the action. By means of the environment readiness variables, the fairness assumptions are stated conditionally. A fairness statement $W/SF : \alpha$ for an action $\alpha(p)$ is translated to the fairness assumption $\forall p : (W/SF(\alpha(p) \wedge p \in e_\alpha))$. So L is translated into

$$L.Init \wedge$$
$$\Box[\exists k : L.fRq(k) \vee \exists k : L.nRq(k) \vee \exists k, d : L.fIn(k, d) \vee \exists k : L.nIn(k)]_{(L.cRq, L.maxIn)}$$
$$\wedge \forall k, d : WF_{(L.cRq, L.maxIn)}(L.fIn(k, d) \wedge [[k, d]] \in L.e_{fIn}).$$

The specification style supports the composition of processes to systems. A composition is defined by a list of process instances and a list of system actions. By style conventions, the system actions are conjunctions of process actions. From each process a process action or a stuttering step has to occur as a conjunct in a system action. If a process action is supplied with a fairness assumption, it has to occur in exactly one system action and

```
process Scomp_flat (usd : datatype ; tc : reflexive relation on (usd, usd) ; ts : nat) ;
 variables
  S.cRq : key ; C.buf : set of [[key, usd]] ; L.cRq, L.maxIn : key ;
 Init ≜ S.Init ∧ C.Init ∧ L.Init ;
 actions
  fRq (krq : key; d : usd) ≜ S.Rq(krq) ∧ C.Rq(krq,d) ∧ L.fRq(krq) ;
  nRq (krq : key; d : usd) ≜ S.Rq(krq) ∧ C.Rq(krq,d) ∧ L.nRq(krq) ;
  fIn (krq : key; d : usd) ≜ S.In(krq) ∧ C.In(krq,d) ∧ L.fIn(krq,d) ;
  nIn (krq : key; d : usd) ≜ S.In(krq) ∧ C.In(krq,d) ∧ L.nIn(krq) ;
 WF : fIn ;
 end ;
```

Figure 3 Example process composition in flat form.

the actual parameters of this occurrence have to correspond to the formal parameters of the system action. The style conventions ensure that a system is equivalent to the logical conjunction of its processes and an additional coupling constraint CC, where CC is an invariant and constrains the occurrence of process actions and the setting of environment readiness variables.

Fig. 2 shows the definition of the process type *Scomp*. A system consists of three processes S, C, and L of the process types defined in Fig. 1. Here, all system actions are three party rendezvous of the three constraint processes. Note, that *Scomp* does not contain explicit fairness assumptions. The fairness assumptions of system actions are derived from the fairness assumptions of process actions. The definition of a composition can be transformed syntactically into the definition of an equivalent process type which we call flat form of the composition. All private variables of the processes are private variables of the flat form. The *Init* condition of the flat form is the conjunction of the *Init* conditions of the processes. The actions of the flat form are the system actions of the composition. If a process action has a fairness assumption, the fairness assumption is inherited to that system action, the process action is contained. The environment readiness variables of process actions are substituted by state functions which refer to the Enabled conditions of its rendezvous partners in system actions. For instance, Fig. 3 shows the flat form of the process type $Scomp$ of Fig. 2. The implicitly declared environment readiness variable $Scomp_{flat}.e_{fIn}$ of the system is assumed to be set by the outside of $Scomp_{flat}$. The environment readiness variable $L.e_{fIn}$ of the component L is constrained by the invariant $\Box(L.e_{fIn} = Scomp_{flat}.e_{fIn} \cap \{[[k,d]] : Enabled(S.In(k) \wedge Enabled(C.In(k,d))\}$. In accordance with the style conventions, this invariant is a part of the coupling constraint CC of *Scomp*. The TLA translation of the system fairness assumption *WF: fIn* is $\forall k, d :$ $WF_{(S.crq,C.buf,L.cRq,L.maxIn)}(Scomp_{flat}.fIn(k,d) \wedge [[k,d]] \in SComp_{flat}.e_{fIn})$.

3 STRUCTURED VERIFICATION

The verification of a protocol specification P implementing a service specification S ($P \Rightarrow S$) can be performed in three different steps (cf. Sect. 1). The step of service decomposition corresponds to reduce S to a set of SCs $SC_1, \ldots, SC_m$ and a coupling constraint SCC guaranteeing the correctness of $SC_1 \wedge \ldots \wedge SC_m \wedge SCC \Rightarrow S$. In the protocol decomposition step P is reduced to a system consisting of a set of PMs $PM_1, \ldots, PM_n$, MCs $MC_1, \ldots, MC_o$, and a coupling constraint PCC. This decomposition is correct if $P \Rightarrow PM_1 \wedge \ldots \wedge PM_n \wedge MC_1 \wedge \ldots \wedge MC_o \wedge PCC$ can be proved. The compositional proof step is performed by proving for each SC_i the implication from a subset of the PMs and MCs $\forall i \in \{1, \ldots, m\} : PM_{i_1} \wedge \ldots \wedge PM_{i_p} \wedge MC_{i_1} \wedge \ldots \wedge MC_{i_q} \wedge P_{S_i}CC \Rightarrow SC_i$. Intermediate layers of processes (eg. FAPMs and APMs) can be used for stepwise refinement proofs. The implication of the coupling constraints ($PCC \Rightarrow SCC$) has to be verified, too. This proof is based on the complexity of the action couplings and moreover gains very much from the SC_i-verifications.

For exemplification, the compositional proof of the SCs *SDUId*, *Corruptions*, and *LiveIn*, introduced in Sect. 2, will be outlined. We use a protocol scenario which consists of two protocol entities S (sender) and R (receiver), an uni-directional channel M between S and R for data transfer, and an uni-directional channel A between R and S for acknowledgement information. The interaction with the protocol user is performed by the actions *Rq* and *In*. The transmission of data to and the reception of data from the channels M and A is performed by the actions *MRq*, *MIn*, *ARq*, and *AIn*. We want to prove, that the SCs are implied by APMs and AMCs which are outlined in Sect. 4. M and A are described by the AMCs *SDUId* and *Corruptions*, which correspond to the SCs except for different process parameter settings. Further, the AMC *Phantoms* and the liveness AMC *LiveInInf* are used. The safety properties of the protocol entities are specified by the APMs *SBufferKey*, *SBufferUsd*, *RBufferKey*, *RBufferUsd*, *SAcknowledge* and *RAcknowledge*. The APMs *SLiveMRq*, *RLiveIn*, and *RLiveARq* are describing liveness properties.

The proof of the SC *SDUId* corresponds to the verification of the implication

$$\begin{array}{l} Let\ Pars \triangleq mtc \subseteq \{(p,q) | spci(p) = spci(q) \vee q \notin encpci\}; \\ \quad Sys \triangleq \ SBufferKey(pdu, pci, encpci, spci, skey) \wedge \\ \qquad\qquad RBufferKey(pdu, pci, encpci, spci, skey, rc, tg) \wedge \\ \qquad\qquad MSDUId \wedge \\ \qquad\qquad MCorruptions(pdu, mtc) \wedge \\ \qquad\qquad MPhantoms(pdu, encpci) \wedge \\ \qquad\qquad CC_{SDUId} \\ in \quad Pars \wedge Sys \Rightarrow SDUId \end{array} \tag{1}$$

The implication is preceded by the keyword *in* and uses the local definitions *Pars* and *Sys* which are defined in the section preceded by the keyword *Let*. In *Sys* the subsystem of APMs, which is sufficient for the proof, is defined. $SBufferKey$ and $RBufferKey$ are APMs, $MSDUId$, $MCorruptions$, and $MPhantoms$ are AMCs. The coupling constraint CC_{SDUId} assumes that the actions of the APM processes with similar names and action parameters are joined. In *Pars* the parameters of the subsystem processes are constrained.

The proof of the SC *Corruptions(stc)* can be performed by verifying formula 2:

$$\begin{array}{l} Let\ Pars \triangleq mtc \subseteq \{(p,q) | (spci(p) = spci(q) \vee q \notin encpci) \wedge ((susd(p), susd(q)) \in stc \vee q \notin encusd)\}; \\ \quad Sys \triangleq \ SBufferKey(pdu, pci, encpci, spci, skey) \wedge \\ \qquad\qquad SBufferUsd(usd, pdu, encusd, susd) \wedge \\ \qquad\qquad RBufferKey(pdu, pci, encpci, spci, skey, rc, tg) \wedge \\ \qquad\qquad RBufferUsd(usd, pdu, encusd, susd) \wedge \\ \qquad\qquad MSDUId \wedge \\ \qquad\qquad MCorruptions(pdu, mtc) \wedge \\ \qquad\qquad MPhantoms(pdu, encpci) \wedge \\ \qquad\qquad CC_{Corruptions} \\ in \quad Pars \wedge Sys \Rightarrow Corruptions(stc) \end{array} \tag{2}$$

$CC_{Corruptions}$ of formula 2 uses a coupling which is similar and compatible to CC_{SDUId}.

$$
\begin{array}{lll}
Let\ Pars \triangleq & atc \subseteq \{(p,q)|spci(p) = spci(q) \vee q \notin encpci\} \wedge & \\
& \forall p \in pdu \exists q \in pdu : spci(p) = spci(q) \wedge susd(p) = susd(q) \wedge q \in encpci \cap encusd \wedge & \\
& \{(p,q)|spci(p) = spci(q) \wedge susd(p) = susd(q)\} \subseteq mla \wedge & \\
& \{(p,q)|spci(p) = spci(q) \wedge susd(p) = susd(q)\} \subseteq ala \wedge & \\
& (c > 1 \vee c = \infty); & \\
\quad Sys \triangleq & SAcknowledge(pdu, pci, encpci, spci, sack) \wedge & \\
& RAcknowledge(pdu, pci, encpci, encusd, spci, sack, rc, tg) \wedge & \\
& SLiveMRq(pdu, pci, encpci, encusd, spci, sack, susd, ts) \wedge & \\
& RLiveIn(pdu, pci, encpci, encusd, rc, tg) \wedge & \\
& RLiveARq(pdu, pci, encpci, encusd, spci, sack, rc, tg) \wedge & \\
& MLiveInInf(usd, mc, mla) \wedge & (3) \\
& ASDUId \wedge & \\
& ACorruptions(pdu, atc) \wedge & \\
& APhantoms(pdu, encpci) \wedge & \\
& ALiveInInf(usd, ac, ala) \wedge & \\
& CC_{LiveIn}; & \\
\quad EnvCond \triangleq & \forall p : Enabled(SLiveMRq.fMRq(p)) \Rightarrow p \in Sys.e_{fMRq(p)} \wedge & \\
& \forall p : Enabled(RLiveIn.fIn(p)) \Rightarrow p \in Sys.e_{fIn(p)} \wedge & \\
& \forall p : Enabled(RLiveARq.fARq(p)) \Rightarrow p \in Sys.e_{fARq(p)} \wedge & \\
& \forall p : Enabled(MLiveInInf.fMIn(p)) \Rightarrow p \in Sys.e_{fMIn(p)} \wedge & \\
& \forall p : Enabled(ALiveInInf.fAIn(p)) \Rightarrow p \in Sys.e_{fAIn(p)} & \\
in \quad Pars \wedge Sys \wedge \Box EnvCond \Rightarrow LiveIn(ts, c) & &
\end{array}
$$

The proof of the liveness SC *LiveIn* corresponds to the verification of formula 3. CC_{LiveIn} assures with regard to joint actions, that all actions with similar names and parameters are joined. The environment condition *EnvCond* constrains the environment of Sys. The environment has to tolerate fair actions whenever fair actions of a liveness process are enabled. Thus, the design of environment conditions depends on the design of liveness processes which is explained in Sect. 6.

After the implication proofs, the verification of the coupling constraint *SCC* is simple. The correctness of the SC action couplings can be directly implied from 1 and 2 since they correspond to the couplings of the APM and AMC actions. The proof of the constraints regarding the environment readiness variables benefit from 3 since we proved the SC *LiveIn* is implied from the APMs and the AMCs.

4 FRAMEWORK ARCHITECTURE

A survey on the SCs, APMs, and FAPMs of the framework will be outlined here. Specifications of these processes are available via World Wide Web (address `http://ls4-www.informatik.uni-dortmund.de/RVS/P-TPM`). Furthermore, the WWW-reference contains the theorems of the framework together with the corresponding proof descriptions.

By the SCs typical safety constraints like absence of data corruptions, losses, reorderings, duplications or phantoms can be specified. Due to different combination of the safety constraints different fairness assumptions are necessary to ensure system liveness. If in a system gaps of received data do not exceed a certain finite limit, it is sufficient to guarantee that a SDU within this limit will eventually be delivered. However, if data losses are not constrained, it must be guaranteed that a SDU will be delivered if it is

Table 1 Service Constraints (SCs)

Safety	SDUId	assuring the unambiguity of the UI (unit identifiers; successive order in *Rq*, extra identifier for phantoms)
	Corruptions(tc)	tolerating data corruptions of user data between sent and delivered data only within a limit (described by the relation tc)
	Phantoms(pd)	preventing phantoms to be of the set pd (subset of user data)
	Gaps(tg)	tolerating gaps of at most tg SDUs (service interface data units) within the sequence of delivered SDUs
	Reorderings(tr)	tolerating reorderings of delivered packets only within a limit (after delivery of a SDU with UI k only packets with UIs $\geq k - tr$ can be delivered)
	Duplicates(td)	limiting the number of duplications of a SDU to at most td deliveries
	Capacity(c)	preventing the number of sent but not delivered SDUs exceeding c
Liveness	LiveIn(ts,c)	guaranteeing the delivery of a SDU if at least $\min(ts+1,c)$ SDUs are in transfer (only compatible to SC *Capacity* if parameter c is identical to parameter c in *Capacity*)
	LiveInInf(c,la)	guaranteeing that continuously sending SDUs of a certain attribute leads to the delivery of at least one SDU (the attributes are defined by the equivalence relation la; only compatible to SC *Capacity* if parameter c is identical to parameter c in *Capacity*)

continuously sent. Therefore two different liveness SCs are defined. All SCs refer to the two actions *Rq* and *In*. Some SCs further define an action *Tick* modelling arbitrary loss events of the service. In Table 1 the SCs are outlined. The parameter *usd* corresponds to the data type of user data units.

Protocol mechanisms for provision of the SCs are modelled by the APMs. Safety APMs for handling and buffering user data and unique identifiers, ordering user data, encoding and decoding PDUs, and checking redundancy of PDUs are currently available. The actions correspond to those described in Sect. 3. Liveness APMs guarantee sending, delivery, and confirmation of user data in dependency of the properties of the underlying service. A survey of essential APMs is shown in Table 2 in which the variable *maxIn* refers to the successor of the largest UI of all SDUs delivered by *In*. Further process parameters are not sketched in Table 2. While user data are described by the process parameter *usd*, the set of PDUs is modelled by *pdu* and the protocol control information (PCI) of a PDU by *pci*. The function *spci* maps a PDU to its part containing the PCI and *susd* to the user data. By *skey* and *sack* a PCI is mapped to the parts containing the UI of a transfered data unit resp. the set of acknowledged UIs. *encpci* and *encusd* are subsets of the PDUs. They can be used for corruption checking if they comprise PDUs whose PCI or user data is accepted by a checking mechanism. The AMCs model the underlying service of the protocol which can be reduced into the two uni-directional channels M and A. The AMCs are similar to the SCs with the process parameter *usd* replaced by *pdu*.

Protocol structures with realistic finite data structures are supported by the FAPMs. This demand causes the necessity of a finite set for modelling UIs. Thus, the unambiguity of the UIs will not be guaranteed anymore and methods for its preservation have to be introduced. These methods are based on finite lifetime assumptions on PDUs in the channels M and A. It is assumed that a PDU will only be delivered if at most *mmem* resp. *amem* PDUs have been sent by *MRq* since its transmission. For M this can be easily modelled by the combination of the SCs *Reorderings(tr)* and *Capacity(c)* assuming

Table 2 Abstract Protocol Mechanisms (APMs)

Safety	SBufferKey	*Rq*: assigning UIs in successive order; *MRq*: generating and sending PDUs from *encpci* whose UI section is either empty or the UI was previously allocated to a SDU
	RBufferKey(*rc*,*tg*)	*In*: only delivery of SDUs with UIs previously received by *MIn*; *MIn*: if a received PDU is a member in *encpci* and the receive buffer limit is not exceeded (determined by *rc* and *tg*), extracting and buffering the UI
	SBufferUsd	*Rq*: buffering a SDU and its UI (assigned by *SBufferKey*); *MRq*: generating and sending PDUs from *encusd* whose user data section is either empty or contains data and UIs which were previously buffered
	RBufferUsd	*In*: delivery of SDUs with buffered user data only; *MIn*: extracting and buffering user data and its corresponding UI (from *RBufferKey*) if the received PDU is member in *encusd*
	SAcknowledge	*MRq*: sending PDUs with UIs only which are not acknowledged; *AIn*: buffering acknowledged UIs if the received PDU is member in *encpci*
	RAcknowledge	*ARq*: generating and sending PDUs from *encpci* whose acknowledgement section contains UIs only which are buffered (cf. *RBufferKey*, *RBufferUsd*) or smaller than *maxIn*
	RPhantoms(*pd*)	*In*: delivery of SDUs, marked to be corrupted or phantoms, only if their user data is not member in *pd*
	RGaps(*tg*)	*In*: delivery of SDUs only whose UIs do not exceed *maxIn*
	RReorderings(*tr*)	*In*: delivery of SDUs only whose UIs are not smaller than *maxIn*
	RDuplicates(*td*)	*In*: delivery of SDUs only which were not already delivered more than *td* times
	SCapacity(*sc*)	*Rq*: enabled only if less than *sc* SDUs are sent but not acknowledged
Liveness	SLiveMRq(*ts*)	*MRq*: the PDU with UI k is guaranteed to be sent eventually if k is the least UI not already acknowledged and at least *ts* SDUs with greater UI were sent by *Rq*
	RLiveIn	*In*: the SDU whose UI equals *maxIn* will eventually delivered if it was received by *MIn*
	RLiveARq	*ARq*: a PDU acknowledgeing all UIs smaller than *maxIn* will eventually be sent

$mmem = tr + c$. For A an additional constraint *ReorderingsMRq* linking *AIn* and *MRq* is used. Two methods for preserving unambiguity are supported. The one uses an interactive management for releasing UIs. Receiving an acknowledgement message of a SDU with UI k, the entity S notifies entity R of the release of k. R acknowledges this message and interprets, after a waiting period of $mmem$, the reception of UI k to be a new SDU. S may assign k after the reception of acknowledgement and a waiting period of $amem$ to a new SDU. By the FAPMs *SRelease* and *RRelease* these protocol mechanisms are described. By the other method no active UI management is provided. We assume that both protocol entities are informed about the maximum buffer sizes of each other and that new PDUs are sent by *MRq* in an incremental order (FAPM *OrderMRq*). Thereupon, we can prove that an UI state space of $ws = 3 * (bs + br) + max(mmem, amem)$ is necessary and sufficient for preserving the unambiguity of SDUs (bs and br are the maximum sizes of the send and receive buffers). On the basis of these methods we can specify the FAPMs similarily to the APMs replacing variables of infinite data types by those of finite types and by using finite buffers.

5 VERIFICATION ELEMENTS

In addition to service and protocol processes, theorems for proving the implication of SCs by APMs and of APMs by FAPMs are provided in the framework. The validity of these theorems were proved. A theorem consists of a subsystem description, of an invariant constraining process parameters, and, in the case of liveness constraints, of an environment condition part.

The proof of liveness processes is discussed in Sect. 6. Three theorems for the proof of SCs are listed in Sect. 3 (formulas 1, 2, and 3). In Table 3 theorems for the proof of SCs by APMs and AMCs are outlined without listing the process parameter invariants. The $*$-symbols in a column mark the APMs and AMCs which contribute to the proof of the SC listed in this column. These theorems are based on AMCs which guarantee the detection of phantoms and corruptions (†) only. The implication theorems for AMCs with more rigorous assumptions are not listed.

The proof of the theorems concerning the verification of APMs by FAPMs can be reduced to two groups according to the method used for preserving unambiguity. The finite PDU lifetime is ensured by using the FAMCs *Capacity*, *Reorderings* for M and, additionally, *ReorderingsMRq* for A. The proof of an APM by its corresponding FAPMs can be performed using either *SRelease* and *RRelease* or using *OrderMRq* and constraining the process parameters regarding the UI state space and the buffer sizes.

Table 3 Implication theorems for the proof of SCs

SCs		SDUId	Corr.	Phant.	Gaps	Reor.	Dupl.	Cap.	LiveIn
APMs	SBufferKey	$*$	$*$						
	RBufferKey	$*$	$*$						
	SBufferUsd		$*$						
	RBufferUsd		$*$						
	SAcknowledge							$*$	$*$
	RAcknowledge							$*$	$*$
	RPhantoms			$*$					
	RGaps				$*$				
	RReorderings					$*$			
	RDuplicates						$*$		
	SCapacity							$*$	
	SLiveMRq								$*$
	RLiveIn								$*$
	RLiveARq								$*$
AMCs	M.SDUId	$*$	$*$						
	M.Corruptions	$*^\dagger$	$*^\dagger$						
	M.Phantoms	$*^\dagger$	$*^\dagger$						
	M.LiveIndInf								$*$
	A.SDUId							$*$	$*$
	A.Corruptions							$*^\dagger$	$*^\dagger$
	A.Phantoms							$*^\dagger$	$*^\dagger$
	A.LiveIndInf								$*$

6 LIVENESS

Environment conditions are essential elements of liveness theorems since they prevent the disturbing of liveness properties of a process by its environment. In order to explain the principles, we refer to the typical scenario of Fig. 4. It contains a set of SCs to be proven by subsystems containing APMs and AMCs. We refer to the proof of the liveness SC SC_1 performed by the verification of the implication $Sys_1 \Rightarrow SC_1$. Sys_1 is a subsystem of APMs and AMCs consisting of the liveness process L and safety processes S_1 and S_2. The proof can be performed if each fair action α of L is never disabled by other processes (e.g., S_1, S_2, S_3, or S_4), i.e., the formula $\Box(\forall p : \mathit{Enabled}(L.\alpha(p)) \Rightarrow p \in L.e_\alpha)$ holds. Because of this, the liveness properties expressed by L are valid within the whole system (e.g., the combination of L, S_1, S_2, S_3, and S_4) as well as within the subsystem Sys_1. The formula is ensured by an environment condition *EnvCond* (for example look at the environment condition of formula 3 in Sect. 3). If *EnvCond* is fulfilled by the processes outside Sys_1 (e.g., S_3, S_4), a fair action α of L cannot be disabled by the environment of Sys_1 (i.e., $\Box(\forall p : \mathit{Enabled}(L.\alpha(p)) \Rightarrow p \in Sys_1.e_\alpha)$ holds). For the co-processes within Sys_1 (e.g., S_1, S_2) we already proved that they do not disable the fair actions of L (e.g., $\Box(\forall p : \mathit{Enabled}(L.\alpha(p)) \Rightarrow (\mathit{Enabled}(S_1.\alpha(p)) \wedge \mathit{Enabled}(S_2.\alpha(p)))))$ holds). Thus, due to the style conventions $\Box(\forall p : \mathit{Enabled}(L.\alpha(p)) \Rightarrow p \in L.e_\alpha)$ can be concluded.

To support modular liveness proofs, we design the liveness processes under consideration of all safety processes. A liveness process L can be coupled with a safety process S. Whenever fair actions of L are enabled in L, the coupled actions of S are enabled, too. Nevertheless, the liveness processes have to guarantee liveness constraints that are sufficient for describing system progress. In the following we first introduce the conventions for specifying liveness processes that constrain the liveness properties but not the safety properties of a system. Afterwards, we discuss the liveness processes of the framework.

In the liveness processes the fair actions are splitted into two actions with complementary enabling conditions (fi., in SC *LiveIn* (cf. Fig. 1) the action *In* is split into *fIn* and *nIn*). In any system state exactly one of the two actions will be enabled. The other actions are always enabled. Therefore liveness processes do not introduce safety constraints. However, by defining only one of the two complementary actions to be fair one can design the enabling conditions of the fair actions to fulfill *EnvCond* (fi. *fIn* in *LiveIn*).

Service specifications (e.g., the SCs) have to fulfill two conditions with regard to liveness. First, the service user must be infinitely often enabled to send data. Second, the ser-

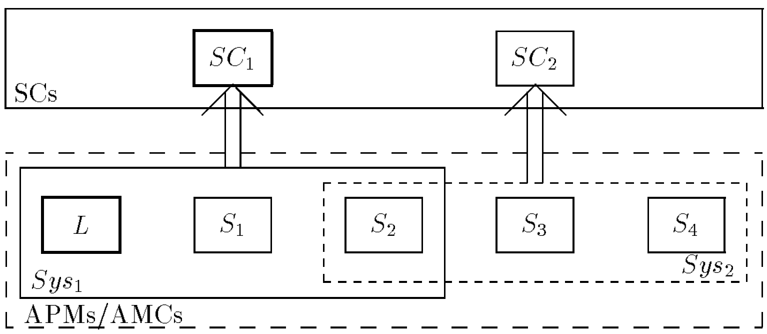

Figure 4 Liveness verification scenario.

vice provider has to react to data requests by delivering data eventually. In the framework the liveness SC *LiveIn* in combination with the safety SC *SDUId* (cf. Fig. 1) is sufficient for guaranteeing these goals. User requests (actions *fRq* or *nRq* in *LiveIn*) are only constrained by the enabling condition of *Rq* in *Capacity*. However, if the process parameters c of *Capacity* and *LiveIn* are equal, the enabling condition of *fRq* in *LiveIn* is not weaker than the one of *Rq* in *Capacity*. In this case *nRq* is never enabled and one can verify that, due to the weak fairness assumption on *fIn*, *fRq* will eventually be continuously enabled until it is executed. The eventual delivery of data depends on the weak fair action fIn which is enabled if $maxIn < cRq + ts$ holds (cRq describes the UI of the SDU transmitted last and $maxIn$ the UI of the SDU delivered last; ts is a process parameter and models tolerance with regard to the number of packets to be send until a delivery is required). Thus, either a SDU with an UI which is greater than $maxIn$ will be delivered or *fIn* will be permanently enabled. In that case, due to its weak fairness, it will eventually be executed performing the delivery of the SDU with UI $maxIn$. Thus, the fairness assumptions on the service provider reaction is fulfilled. These liveness assumptions can be implemented by protocol entities which contain relatively simple fairness assumptions. With regard to the APMs and FAPMs, it is sufficient that the entity S lively transmits the SDU with the smallest UI which was not acknowledged. Entity R is obliged to deliver the SDU with UI $maxIn$ eventually if it is in the receive buffer and to acknowledge all SDUs with UI being smaller than $maxIn$ (cf. the APMs *LiveMRq*, *LiveIn*, and *LiveARq* in Table 2). These weak liveness assumptions of the SCs as well as of the APMs and FAPMs facilitate the definition of liveness processes which fulfill the environment liveness conditions for each combination of safety processes (the formula $SafetyProc \wedge LivenessProc \wedge CC \Rightarrow \Box EnvCond$ holds for every safety process $SafetyProc$). In the framework, the only limitations concern the process parameters of the SCs, APMs, and FAPMs constraining the system capacity (fi. $Capacity(c_1) \wedge LiveIn(c_2) \Rightarrow \Box EnvCond \vee (c_1 < c_2)$).

7 CONCLUDING REMARKS

While the benefits of formally proven designs are commonly recognized, formal protocol verification is not yet widely applied in practical protocol and communication system developments. Besides of the necessary education and training in the techniques, a major problem for the practical 'daily-life' application is the additional effort for formal modelling and reasoning. With respect to this, existing approaches mainly concentrate on two topics, on the tool-support and on the general fundamentals of suitable formal modelling and verification techniques. Our paper pleads for the complementary consideration of a third topic, the support of users by suitable application-specific theories. They can help to bridge the gap between theory and practice. The re-usable elements of the theories facilitate the formal modelling and reasoning tasks enormously. Moreover they can directly support the systematic understanding of application-field specific problems and methods. That this is an actual problem, we can conclude e.g. from the NewCoRe project (Holzmann, 1994). G. Holzmann reports that almost 55% of the original design specifications were logically inconsistent. This experience is a weighty argument for the direction of research we pursue in our work.

8 REFERENCES

Abadi, M. and Lamport, L. (1991) The Existence of Refinement Mappings. *Theoretical Computer Science*, **82(2)**, 253–84.

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

Clark, D.D. and Tennenhouse, D.L. (1990) Architectural Considerations for a New Generation of Protocols. *Computer Communications Review*, **20(4)**, 200–8, ACM SIGCOMM'90 Symposium, Philadelphia.

Doeringer, W.A., Dykeman, D., Kaiserswerth, M., Meister, W., Rudin, H., and Williamson, R. (1990) A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communications*, **38(11)**, 2025–39.

Haas, Z. (1991) A Protocol Structure for High-Speed Communication over Broadband ISDN. *IEEE Network Magazine*, Jan., 64–70.

Herrmann, P. and Krumm, H. (1994) Compositional Specification and Verification of High-Speed Transfer Protocols, in *Protocol Specification, Testing, and Verification XIV* (ed. S.T. Vuong and S.T. Chanson), IFIP, Chapmann & Hall, 339–46.

Holzmann, G.J. (1990) Algorithms for Automated Protocol Verification. *AT&T Technical Journal*, Jan., 32–44.

Holzmann, G.J. (1994) The Theory and Practice of A Formal Method: NewCoRe, in *13th World Computer Congress 94*, volume I (ed. B. Pehrson and I. Simon), IFIP, Elsevier.

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

Kurki-Suonio, R. and Järvinen, H.M. (1988) Action system approach to the specification and design of distributed systems. *ACM Transactions on Programming Languages and Systems*, **4(10)**, 510–54.

La Porta, T.F. and Schwartz, M. (1993) The Multistream protocol: A highly flexible high-speed transport protocol. *IEEE Journ. on Sel. Areas in Communications*, **11(4)**.

Lamport, L. (1992) TLA+: Syntax and Semantics, preliminary version, Research Report, DEC Digital Systems Research Center, Palo Alto, to appear.

Lamport, L. (1994) The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, **16(3)**, 872–923.

Mester, A. and Krumm, H. (1995) Composition and Refinement Mapping based Construction of Distributed Applications, in *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TAPSOFT'95 satellite)*, BRICS, to appear.

O'Malley, S.W. and Peterson, L.L. (1991) A Highly-Layered Architecture for High-Speed Networks, in *Protocols for High-Speed Networks, II* (ed. M.J. Johnson), IFIP WG 6.1/WG 6.4, North-Holland.

Plagemann, T., Plattner, B., Vogt, M., and Walter. T. (1993) Modules as Building Blocks for Protocol Configuration. in *International Conference on Network Protocols, Los Alamitos, California*, IEEE Computer Society Press.

Shankar, A.U. and Lam, S.S. (1992) A Stepwise Refinement Heuristic for Protocol Construction. *ACM Transactions of Programming Languages and Systems*, **14(3)**, 417–61.

Vissers, C.A., Scollo, G., and van Sinderen, M. (1988) Architecture and specification style in formal descriptions of distributed systems, in *Protocol Specification, Testing and Verification VIII* (ed. S. Agarwal and K. Sabnani), IFIP, Elesevier, 189–204.

Zitterbart, M., Stiller B., and Tantawy, A.N. Application-Driven Flexible Protocol Configuration, in *Kommunikation in Verteilten Systemen* (ed. N. Gerner, H.-G. Hegering, and J. Swoboda), GI/ITG, 8. Fachtagung, München, Springer-Verlag, 144–58.