A Framework for Modeling Transfer Protocols

Peter Herrmann, Heiko Krumm
Universität Dortmund, Fachbereich Informatik
D-44221 Dortmund
Internet: {herrmann|krumm}@ls4.cs.uni-dortmund.de
Phone: +49-231-7554674, Fax: +49-231-7554730

Abstract

The notion of specification frameworks transposes the framework approach from software development to the level of formal modeling and analysis. A specification framework is devoted to a special application domain. It supplies re-usable specification modules and guides the construction of specifications. Moreover, it provides theorems to be used as building blocks of verifications. By means of a suitable framework, specification and verification tasks can be reduced to the selection, parametrization and combination of framework elements resulting in a substantial support which opens formal analysis even for real-sized problems. The transfer protocol framework addressed here is devoted to the design of data transfer protocols. Specifications of used and provided communication services as well as protocol specifications can be composed from its specification modules. The theorems correspond to the relations between protocol mechanism combinations and those properties of the provided service which are implemented by them. This article centers on the application of this framework which is discussed with the help of the specification of a sliding window protocol. Moreover the structure of its verification is described. The specification and verification technique applied is based on L. Lamport's Temporal Logic of Actions (TLA). We use the variant cTLA which particularly supports the modeling of process systems.

Keywords: Protocol specification, Protocol verification, Temporal logic, Framework, Protocol composition

1 Introduction

Due to recent developments in the field of high-speed and multimedia communication many new data transfer protocols and protocol variants are designed. Although standardized formal description techniques (i.e., ISO/OSI: Estelle [30] and Lotos [31], ITU: SDL [32]) are available, the protocols are developed frequently without any formal support. Furthermore, abstract service specifications, describing the relevant properties of the communication service provided by a protocol, are often omitted. Thus, the docu-

mentation of a protocol development is often incomplete and ambiguous. Therefore the correctness of the protocol cannot be checked systematically.

On the one hand, ambiguous documentations cause the danger of design errors due to misunderstandings. This might lead to reworking efforts, project delays, and, consequently, to higher project costs. On the other hand, the usage of formal techniques in protocol design causes significant expense, too. While sometimes the application of these techniques can be supported by tools (e.g., special editors, interpreters, compilers, and verification tools; cf. [3, 4, 9, 12, 13, 14, 29, 33, 40]), formal models and formal descriptions of protocols have to be developed in a creative manner. Due to the complexity of modern high-speed protocol systems this task can be quite expensive. Furthermore, the development of formal descriptions is prone to errors and, similar to program development, a long debugging phase may be necessary before completing the verification successfully.

The transfer protocol framework [21, 22] facilitates the development of formal specifications similarly to program development support by libraries of reusable program modules. The framework contains specification modules which can be instantiated and combined to a protocol or service specification. A protocol specification consists of a set of modules, each modeling a single protocol mechanism (e.g., sequence numbering of protocol data units, repeat request, time out). The developer does not need to create a protocol mechanism description from scratch. Instead, he uses a module of the framework and concentrates on suitable instantiations of parameters and on the combination of different modules. Thus, he models the logical structure of the protocol directly, facilitating the understanding of the protocol component co-operation. Furthermore, the developer creates a service specification as well. This task is supported by the framework, too, which contains specification modules modeling single constraints of services (e.g., no corruptions of transfered data, liveness of transfered data, no phantoms). The developer parametrizes and combines these modules to a service specification describing the properties of the service to be provided by the protocol.

The transfer protocol framework supplies a comprehensive collection of specification modules. It comprises each typical functional property of data transfer services as well as all of the protocol mechanisms we found in present data transfer protocols (cf. [8, 35]). Moreover, it is based on few basic assumptions only. Therefore the framework support is very general and it can complement other approaches which focus directly on problem-oriented design, construction, and modular implementation of new transfer protocols. So, modern high-speed data transfer protocols like XTP [44] and MSP [36] can efficiently be specified by means of the framework [24, 27]. Moreover, the framework may be used for the formal modeling and analysis of those protocol configurations which are in the range of current dynamic communication system configuration approaches like DaCapo [41], F-CCS [45], and AVOCA [39]. Due to the general orientation of the framework, it should even cover the formal modeling of most future transfer protocols. Nevertheless extensions are possible and can easily be introduced by addition of new modules. Furthermore, the flexibility of the specification modules contributes to the broad applicability of the framework. Each module corresponds to a generic behaviour type where generic module parameters support the adaption of module instances to special contexts and requirements. With respect to this, specification modules may be viewed as behaviour patterns. Relations exist to approaches which explicitly transpose the notions of design and software

patterns (cf. [11]) to the development of formal specifications (e.g., [15, 38]). Additionally, there is further work focussing on the pattern-based and tool-assisted refinement of abstract cTLA specifications into detailed software specifications [37] and corresponding implementation-oriented framework extensions may be of interest. In the sequel, however, we do not want to stress this point in order to concentrate on practical aspects of framework-based protocol specification and verification.

Formal protocol verification is based on separate specifications of the protocol and the service to be provided. It proves that the protocol actually provides the service and is an effective means for the detection of design errors. It can be performed mechanically by tools based on reachability graph analysis (e.g., state space exploration [28], model checking [6, 7, 10, 16, 20, 29]) if the number of reachable system states of the protocol model is relatively small. However, the state space of most protocols relevant in practice exceeds the limitations of automated tools. Thus, either the protocol specification has to be simplified, or the verification has to be performed by symbolic logical reasoning. Both, the design of suitable simplifications as well as the user-guided computation of logical proofs, increase the protocol development costs significantly.

With respect to this, the verification can be facilitated substantially by the framework theorems taking into consideration that both the service and the protocol specifications consist of existing framework modules. Usually, each service constraint is implemented by a special protocol subsystem combining those protocol mechanisms which ensure that the protocol execution complies with the service constraint. For each possible pair of a service constraint and protocol subsystem we verified a theorem stating that the protocol subsystem implies the service constraint. Therefore, a subsystem of the protocol specification as well as a corresponding theorem exists for each constraint of the service specification. Thus, the protocol verification can be accomplished by identifying suitable "service constraint – protocol subsystem – theorem" triples. Additionally, the developer has to check that the protocol mechanism specifications of the subsystem and the service constraint specification are parametrized in a suitable and consistent way. By this method even complex transfer protocols can be verified quite easily [24]. Moreover, this verification method emphasizes the logical relations between protocol subsystems and service components supporting the understanding of the designed protocol.

Thus, framework-based protocol verifications are relatively easily to develop although they rely on formal proofs. Nevertheless, we have presently to point to some reservations concerning the stringency of verifications. They result from the way, the framework theorems were proved. We used the cTLA-tool cTc [26] to compute the process compositions of the theorems. Then we designed the necessary refinement mappings and invariants in order to prepare the manual TLA-based proofs, which — due to the high efforts needed — were not all performed in full formal detail. So, complete proof documentations exist only for some theorems (see [17]). Moreover we did not check the proofs by means of theorem prover tools. Therefore we do not recommmend the use of the theorems for high risk applications though we are convinced of the over-all validity of the theorems.

For the proofs of the framework theorems high efforts were needed and they would substantially profit from the use of theorem prover tools. The framework application, however, is based on the design of suitable theorem instances which is accompanied by relatively simple parameter checks. Consequently, we made the experience that the trans-

fer protocol framework can successfully be applied without special tool-assistance. Nevertheless, one can provide tool-support concentrating on the analysis of compositional specification structures, the search of appropriate theorems, the proposal of theorem instances, and the preparation of predicate logic based parameter checks. A corresponding tool was developed and is described in [18].

Since a large number of theorems would be necessary to link all suitable protocol mechanism combinations with service constraints, the framework supports protocol proofs which are performed in two steps. Therefore, it provides a third collection of specification modules in addition to service constraints and protocol mechanisms. These modules, called abstract protocol mechanisms, support specifications which are on an intermediate abstraction level between protocol specifications and service specifications. Consequently, the theorems do not state directly that protocol mechanisms implement service constraints. Instead, the framework contains two collections of theorems. The theorems of one collection state that protocol mechanism combinations implement abstract protocol mechanisms. The other theorems express that abstract protocol mechanism combinations implement service constraints.

The reduction of protocol verifications into a series of subsystem proofs is based on a special form of compositionality, the superposition. It guarantees that a relevant property of a subsystem is also a property of the system as a whole. Superposition was introduced in [5] as a helpful means for the formal design of systems. Likewise, [2] proposed a transition system based specification technique supporting the formal design of distributed systems by superposition. Our approach applies the specification technique cTLA [21, 37] which is based on TLA (Temporal Logic of Actions, [34]). cTLA supports the modular definition of generic process types and the composition of process systems. Similarly to the standardized formal description technique Lotos [31] (and similarly to [2]), the processes of a cTLA system interact via synchronous joint actions. The process composition operation of cTLA corresponds to the logical conjunction of processes. Therefore, it covers superposition with respect to all relevant safety and liveness properties. The processes of a cTLA system can model logical behaviour constraints as well as components of implementations (cf. constraint-oriented and ressource-oriented specification styles [43]).

In the sequel we concentrate on the practical application of the framework. Therefore, we outline cTLA, its semantics, and the transfer protocol framework concisely. Thereafter, an example application shall clarify the utilization of the framework in more detail. We describe the development of the service and protocol specifications for a well known sliding window protocol (from [42]). Moreover, we discuss the verification of this protocol and point out how the structure of the specifications guides the verification.

2 cTLA

cTLA systems are composed of processes. A process is modeled by a state transition system whose structure is defined by a process type in a programming language-like syntax. As an example we outline the definition of the process type *Corruptions* in Fig. 1. The header consists of the keyword `PROCESS`, the process type name *Corruptions*, and a list of generic parameters (i.e., the symbols `usd` and `tc`). `usd` models the set of data units

```
PROCESS Corruptions ( usd : Any ; ! usd : set of user data transfered
                      tc :  SUBSET(usd × usd) )
                                  ! tc : relation of tolerated corruptions
IMPORT Symbols;
BODY
  VARIABLES
    buf : SUBSET(key × usd);     ! Buffer of all data units ever sent
  INIT ≜ buf = ∅;
  ACTIONS
    Rq (krq : key; d : usd) ≜  ! Transmission of user data d with seq. no. krq
      buf′ = buf ∪ {(krq,d)} ;
    In (krq : key; d : usd) ≜  ! Delivery of user data d with seq. no. krq
      ( krq = "notsent" ∨
            ! Service provider markeda  phantom with a special key "notsent"
        ∀ e ∈ usd :: ((krq,e) ∉ buf) ∨
            ! Phantoms may be delivered without special mark
        ∃ e ∈ usd :: ((krq,e) ∈ buf ∧ (e,d) ∈ tc ) ) ∧
            ! Submitted data may only be delivered if it is corrupted
            !  within limitations set by tc
      buf′ = buf ;
  END
```

Figure 1: cTLA safety process type *Corruptions*

transfered between two service users while the relation `tc` describes a set of corruptions which can be tolerated. In process instances specifying that corruptions are not tolerated at all, `tc` is instantiated with the identity relation. The construct `IMPORT` refers to the inclusion of other modules containing definitions of symbols (fi., data types, functions, and constants). We assume that the symbol `key`, which specifies the set of sequence numbers for data units, is defined in the module *Symbols*. The state space of a process is modeled by variables declared in the section `VARIABLES`. In the process type *Corruptions* `buf` is the only variable. It describes a set of pairs of a sequence number and a user data unit. A predicate headed by the keyword `INIT` specifies the set of initial states of a process. In *Corruptions* the variable `buf` equals to the empty set in the initial state. The action definition part is headed by the keyword `ACTIONS`. An action is a predicate about a pair of a current state and a next state, modeling a set of state transitions. The current state is referenced by variables (fi. `buf`) while the next state is referenced by so-called primed variables (fi. `buf'`). Each pair of a current state and a next state satisfying the predicate is a state transition corresponding to an occurrence of the action. Action definitions can contain data parameters. In the example the action `Rq(2,"data")` applies to all transitions in which the variable `buf` in the next state is equal to `buf` in the current state extended by the pair `(2,"data")`. The variables, `INIT`, and the actions of a process define a state transition system describing a set of state sequences. A state sequence models a possible behaviour of the process if the first state fulfills `INIT` and each state transition corresponds to an action of the process.

The example process is a service specification module of the framework modeling the constraint that the service transmits data without corruptions. The action `Rq` specifies the submission of data to the service and the action `In` the delivery of transmitted data to the service user. The variable `buf` corresponds to the set of all data units ever submitted. Thus, the term ∃ `e` ∈ `usd` :: ((`krq`,`e`) ∈ `buf` ∨ (`e`,`d`) ∈ `tc`) in the definition of the action `In` models that a data unit submitted before (action `Rq`) may only be delivered to the service user (action `In`) if it was corrupted only within the limits defined by the relation `tc`. According to the distinction between safety and liveness properties (cf. [1]), the process type *Corruptions* specifies only safety properties. Therefore *Corruptions* does not rule out state sequences which contain only a finite number of state changes (i.e., processes terminating eventually are tolerated).

```
PROCESS LiveInNoAttr
IMPORT Symbols;
BODY
VARIABLES
    cRq : key ;          ! Sequence number of next data unit to be sent
    maxIn : key ;        ! Sequence number of next data unit to be delivered
INIT ≜ cRq = 0 ∧ maxIn = 0 ;
ACTIONS
    Rq ≜                 ! Transmission of user data
      cRq′ = cRq + 1 ∧ maxIn′ = maxIn;
    fIn ( krq : key ) ≜  ! Delivery of user data with seq. no. krq
                         ! Weak fairness assumed
      krq ≠ "notsent" ∧ krq = maxIn ∧  maxIn < cRq ∧
      maxIn′ = maxIn + 1 ∧ cRq′ = cRq;
    nIn ( krq : key ) ≜  ! Delivery of user data with seq. no. krq
                         ! No fairness assumed
      not (krq ≠ "notsent" ∧ krq = maxIn ∧ maxIn < cRq) ∧
      maxIn′ = IF (krq = "notsent") THEN maxIn ELSE max(maxIn,krq + 1) ∧
      cRq′ = cRq;
  WF: fIn;
END
```

Figure 2: cTLA liveness process type *LiveInNoAttr*

The process type *LiveInNoAttr* in Fig. 2 is an example of a framework module modeling a liveness property. In comparison to the process type *Corruptions*, listed in Fig. 1, *LiveInNoAttr* contains a new construct `WF`. While the other parts of *LiveInNoAttr* define a state transition system again, `WF` describes that the action `fIn` has to be performed "weak-fairly". Similarily, by the construct `SF` one can declare an action to be performed "strong-fairly". A weak fair action must be performed eventually if otherwise it would be enabled continuously for an infinite period of time. A strong-fair action has to be performed even if the action is disabled from time to time. Weak and strong fair actions were introduced in [1]. In contrast to direct liveness properties these fairness assumptions do not model implicit safety properties which might be in contrast to the (explicit) safety properties of the process. Therefore in TLA [34] and cTLA — especially in the transfer

protocol framework — liveness properties are modeled by weak and strong fair actions only.

To support the modularity of the transfer protocol framework, the fairness assumptions should be as weak as possible. Therefore actions are split into two actions, of which one is fair (e.g., `fIn` in process type *LiveInNoAttr*). The other action (e.g., `nIn` in *LiveInNoAttr*) is not fair. Both `fIn` and `nIn` in *LiveInNoAttr* model the delivery of transfered user data (compare action `In` in process type *Corruptions*). The fair action `fIn` specifies that the data unit expected next in the order of transfered data will be delivered lively. Since the process type *LiveInNoAttr* shall concentrate on liveness and therefore must not constrain the delivery of data in general, it contains another action *nIn* tolerating the delivery of other data units.

```
PROCESS SlidWindService (usd : any )  ! usd : set of user data transfered

  PROCESSES
    Id : SDUId;                 ! Assignment of unambiguous sequence numbers
    C : Corruptions (usd,{ (k,k) | k ∈ usd });
                                ! No Corruptions of transfered data
    G : Gaps (0);               ! No Gaps in transfered data stream
    R : Reorderings (0);        ! No reorderings in transfered data stream
    D : Duplicates (0);         ! No duplications of transfered data
    P : Phantoms (usd, usd);    ! No phantoms
    Cap : Capacity (8);         ! Service capacity of eight data units
    LIn : LiveInNoAttr;         ! Data Units are delivered lively

  ACTIONS
    Rq (krq : key; d : usd) ≜   ! Transmission of user data d with seq. no. krq
      Id.Rq(krq) ∧ C.Rq(krq,d) ∧ G.stutter ∧ R.stutter ∧ D.stutter ∧
      P.stutter ∧ Cap.Rq(krq) ∧ LIn.Rq;
    fIn (krq : key; d : usd) ≜  ! Delivery of user data d with seq. no. krq
                                ! Weak fairness assumed
      Id.In(krq) ∧ C.In(krq,d) ∧ G.In(krq) ∧ R.In(krq) ∧ D.In(krq) ∧
      P.In(krq,d) ∧ Cap.In(krq) ∧ LIn.fIn(krq);
    nIn (krq : key; d : usd) ≜  ! Delivery of user data d with seq. no. krq
                                ! No fairness assumed
      Id.In(krq) ∧ C.In(krq,d) ∧ G.In(krq) ∧ R.In(krq) ∧ D.In(krq) ∧
      P.In(krq,d) ∧ Cap.In(krq) ∧ LIn.nIn(krq);

END
```

Figure 3: Service specification *SlidWindService*

Similarly to Lotos [31], cTLA supports the composition of systems from processes. The processes interact via synchronous joint actions. Action parameters model the communication of data. The variables of a process are private and therefore cannot be accessed by other processes. Like each process, the system as a whole is a state transition system. The vector of all variables of all processes forms the system state. A system behaviour is

a sequence of system states where the state changes correspond to system actions. Each system action is defined by the logical conjunction of process actions. In this conjunction, each process is represented either by a true process action or by the pseudo action "stutter" denoting that the process does not perform a state change in this system action.

The specification *SlidWindService*, listed in Fig. 3, models such a system composed of processes. In the part headed by `PROCESSES` the processes of the system are declared. For example, the process *Id* is an instance of the process type *SDUId* and the process C is an instance of the process type *Corruptions*, described above, with the parameter setting $(\mathtt{usd}, \{(k,k) | k \in \mathtt{usd}\})$. In the `ACTIONS` part the system actions are declared by conjunctions of process actions. For instance, the local process actions `Rq` of the processes *Id*, *C*, *Cap*, and *LIn* are coupled to the system action `Rq` while the processes G, R, D, and P participate by stuttering steps only.

The specification technique cTLA supports superposition (cf. [2, 5]) which guarantees that a property fulfilled by a process or subsystem is also a property of each system containing this process or subsystem. It is essential for the conception of the framework and, particularly, for structuring the verification into subsystem implications. With regard to the safety properties, which constrain initial states and state transitions of systems only, superposition can be guaranteed quite easily. Since a system state is modeled by a vector of process states which are defined by private variables only, properties constraining the states of a single process also constrain the states of the whole system.

With respect to liveness properties, however, superposition is more subtle. In a system, a process action can be coupled with actions of other processes. Thus, the environment of a process can block the process action, and due to the blocking the fairness assumption of this process action may be violated. In contrast to TLA, which adopts the fairness assumptions introduced in [1] directly, cTLA therefore uses conditional fairness assumptions only. The WF/SF constructs of cTLA refer to periods of time where a process action is enabled as well as the action is not blocked by the environment of the process. For instance, the process $LIn : LiveInNoAttr$ (Fig. 2) only has the liveness property that data submitted (action `Rq`) will be eventually delivered (actions `fIn` or `nIn`) if the process action `fIn` is not blocked by the environment of *Lin* too often.

On the one hand, this restriction of liveness properties to conditional fairness directly supports superposition. On the other hand, however, conditional fairness assumptions are not able to express absolute liveness properties which are of particular interest for the system design. Nevertheless, one can specify absolute liveness properties by means of an additional condition preventing the process environment to block fair actions too often. If one can prove the safety property that the fair process action `fIn` is tolerated by the process environment whenever it is enabled due to the local process states in *LiveInNoAttr*, the conditional fairness corresponds to the unconditional fairness according to [1]. All modules of the framework, which describe liveness properties correspond to the pattern of the example module *LiveInNoAttr* and describe fairness assumptions as weak as possible. They separate fair subactions (e.g., `fIn`) and are designed to express absolute liveness properties under the assumption that the fair subactions are only blocked by the environment in system states where they are disabled at all. Indeed, the modules of the framework are designed under consideration of each other in order to fulfill this assumption.

3 Formal Semantics

The language cTLA supports the modular formal specification of distributed systems and applies state transition system based modeling like the standard languages Estelle [30] and SDL [32]. Nevertheless, these standard languages mainly concentrate on the structured and easy-to-read formal description of systems and accordingly provide a rich set of language constructs. Formal verifications, however, need additional means since the languages do not directly support formal reasoning. In contrast to these standard languages cTLA has been designed under the objective of explicit verification support. Therefore it is directly based on L. Lamports temporal logic of actions TLA [34]. Thus, each cTLA process instance corresponds to a TLA formula. This correspondence defines the formal semantics of cTLA. Moreover, it enables TLA-based verification.

In TLA canonical formulas describe the safety and liveness properties of state transition systems in accordance to [1]. Inference rules support the syntactical deduction of valid formulas. Moreover, there is an interesting conception of refinement. A more detailed specification *Im* correctly refines a more abstract specification *Sp* if the implication $Im \Rightarrow \overline{Sp}$ is a valid TLA-formula where $\overline{Sp}$ is the specification *Sp* under substitution of its free state variable occurrences by functions of the state variables of *Im*. These functions form the so-called refinement mapping. Due to the correct refinement *Im* implements *Sp* in the practically relevant sense that *Im* meets the safety and liveness requirements expressed by *Sp*. cTLA adopts these notions and techniques of TLA. It is an extension of TLA adding explicit notions of processes, process types, and process composition (as mentioned, the superposition character of composition is of particular interest). Furthermore, there is a different look of cTLA specifications since in cTLA canonical parts of formulas are not explicitly written down.

The TLA formula which corresponds to a process instance depends on the parameter settings of the instantiation and on the definition of the process type. Process type parameters are generic. As usual, we assume that the text strings of actual parameters replace the formal parameter occurrences in the process type definitions. With respect to the process type definitions, there are two forms, one for simple processes and one for systems. In both cases, the TLA formulas of process instances are in canonical form and refer directly to predicate and action definitions of the process type.

For clarification of simple processes let $LIn : LiveInNoAttr$ be a process instance of process type *LiveInNoAttr* listed in Fig. 2. *LIn* corresponds to following TLA formula $\widehat{LIn}$ where the symbols `INIT`, `Rq`, `fIn`, and `nIn` represent the initial state predicate and actions of the process type *LiveInNoAttr*:

$$\begin{aligned}\widehat{LIn} \triangleq\ & \mathtt{INIT} \wedge \\ & \Box[\ \mathtt{Rq} \vee \\ & \quad \exists krq \in \mathtt{key} :: \mathtt{fIn}(krq) \vee \\ & \quad \exists krq \in \mathtt{key} :: \mathtt{nIn}(krq)]_{(\mathtt{cRq},\mathtt{maxIn})} \wedge \\ & \forall krq \in \mathtt{key} :: WF_{(\mathtt{cRq},\mathtt{maxIn})}(\mathtt{fIn}(krq) \wedge krq \in e_{fIn})\end{aligned}$$

$\widehat{LIn}$ is an usual canonical TLA formula describing the safety and liveness properties of a state transition system with the two state variables `cRq` and `maxIn`. The first four lines are devoted to safety properties and define the initial states and the next state

relation of the state transition system. The last line addresses liveness and adds fairness assumptions on subrelations of the next state relation. There is only one particularity in formula $\widehat{LIn}$. While usually fairness operators refer directly to actions of the next state relation, here the weak fairness is not applied to the action `fIn` but to a subaction of it, namely to the conditional action `fIn(krq)` $\wedge$ `krq` $\in e_{fIn}$, where an additional state variable e_{fIn} is addressed. We assume that for each fair action of a process a corresponding environment readiness variable like e_{fIn} exists. The variable is shared between the process and its environment. It is written by the environment and read by the process. It acts as an abstraction of the process environment and indicates the current readiness of the environment for the action. The value of the variable is the set of those action parameter values for which the environment currently does not block occurrences of the action. For instance, when e_{fIn} is empty, the environment is assumed to block the action `fIn` under each possible parametrization. In contrast, the TLA formula $\widehat{LIn} \wedge \Box e_{fIn} = \texttt{key}$ where `key` denotes the set of all possible values for the action parameter `krq` describes a state transition system modeling the process *LIn* in an environment which always tolerates state transitions according to `fIn`.

The TLA formula of systems shall be explained by an extension of the example. We refer to the process type *SlidWindService* which is defined in Fig. 3. As explained in Sec. 2, instances of this type are systems which are composed from eight process instances *Id*, ..., *LIn*. Since all other processes besides *LIn* do not express liveness properties, fairness assumptions only apply to actions of *LIn* (as shown in Fig. 2 the action `LIn.fIn` is accompanied by a fairness statement). Let *SWS* : *SlidWindService(usd)* be an instance of this system type. It corresponds to following canonical TLA formula $\widehat{SWS}$:

$$
\begin{aligned}
\widehat{SWS} \triangleq\ & \texttt{Id.INIT} \wedge \texttt{C.INIT} \wedge \texttt{G.INIT} \wedge \texttt{R.INIT} \wedge \\
& \texttt{D.INIT} \wedge \texttt{P.INIT} \wedge \texttt{Cap.INIT} \wedge \texttt{LIn.INIT} \wedge \\
& \Box[\ \exists krq \in \texttt{key}, d \in usd :: \texttt{Rq}(krq, d) \vee \\
& \quad \exists krq \in \texttt{key}, d \in usd :: \texttt{fIn}(krq, d) \vee \\
& \quad \exists krq \in \texttt{key}, d \in usd :: \texttt{nIn}(krq, d)]_{(\texttt{Id.cRq}, \texttt{C.buf}, \ldots, \texttt{LIn.cRq}, \texttt{LIn.maxIn})} \wedge \\
& \forall krq \in \texttt{key} :: \\
& \quad WF_{(\texttt{Id.cRq}, \texttt{C.buf}, \ldots, \texttt{LIn.maxIn})}(\exists d \in usd :: \texttt{fIn}(krq, d) \wedge krq \in se_{fIn})
\end{aligned}
$$

The formula $\widehat{SWS}$ refers to the definitions of the initial predicates of the process instances. Thus, `LIn.INIT` stands for the initial predicate of *LIn*. Moreover, $\widehat{SWS}$ refers to the action definitions of process type *SlidWindService* which are conjunctions of process actions. For instance, as listed in Fig. 3, `fIn(krq,d)` is defined as `Id.In(krq)` $\wedge \ldots \wedge$ `LIn.fIn(krq)`. Since the action `fIn` of *SWS* contains the fair process action `LIn.fIn`, it also is accompanied by a fairness statement. The last line of $\widehat{SWS}$ states a corresponding fairness assumption for the system action `fIn`. As in simple processes the fairness assumption is conditional and the symbol se_{fIn} denotes the corresponding environment readiness variable for the system action `fIn`.

For the reasoning on process compositions not only TLA formulas of the form of $\widehat{SWS}$ are used. Moreover a compositional form exists which is a conjunction of the TLA

formulas of the constituting processes. Thus, the compositional formula $\widetilde{SWS}$ of *SWS* is:

$$\widetilde{SWS} \triangleq \widehat{Id} \wedge \widehat{C} \wedge \ldots \wedge \widehat{LIn} \wedge SCC \wedge$$
$$\Box(\texttt{LIn}.e_{fIn} = \{krq \in \texttt{key} | \exists d \in usd :: krq \in se_{fIn} \wedge Enabled(\texttt{Id.In}(krq)) \wedge \ldots \wedge Enabled(\texttt{Cap.In}(krq))\})$$

In addition to the formulas of the constituting processes the compositional formula $\widetilde{SWS}$ conjoins two invariants. The so-called coupling constraint SCC expresses the safety conditions which result from the special action coupling of a system, e.g., it states with respect to the action `LIn.fIn(krq)` that this action has to occur in combination with simultaneous occurrences of the actions `Id.In(krq)`, `C.In(krq,d)`, ..., `Cap.In(krq)`. The last invariant defines the environment readiness variable $LIn.e_{fIn}$ of process *LIn* as state function: The action `fIn` of process *LIn* is tolerated by *LIn*'s environment for a parameter value `krq`, exactly if the environment of *SWS* tolerates the system action `fIn` and all these actions of the other processes of *SWS* are enabled which are coupled with `LIn.fIn` within the system action `fIn`.

Besides of the syntax and the TLA transformation rules outlined above, the cTLA language definition contains few conditions restricting the content of system action definitions especially in context with the occurrence of fair actions. In the main, these conditions express the assumption, that fair process actions are disjoint and that for each fair process action there is exactly one containing system action. Under these conditions one can prove that the compositional TLA formula of a system instance is equivalent to the direct canonical TLA formula of this system instance (the proof is described in [17]). In the example this means that $\widehat{SWS} \Leftrightarrow \widetilde{SWS}$ is valid.

From the equivalence of the compositional formula with the direct canonical formula of a system instance we can infer that the compositional formula is free from contradiction. In combination with the form of the compositional formula which conjoins the formulas of the constituting processes, we can infer that process composition implies the consistent logical conjunction of processes. Therefore a system formula implies the formula of each constituting process. This means, that process composition has ideal superposition character because the safety and liveness properties of processes are also properties of containing systems. Moreover, since logical conjunction is commutative and associative, superposition applies not only to processes but also to subsystems of a system.

Superposition facilitates the formal verification of system properties. In order to prove that a system S has the properties expressed by a TLA formula P, it is sufficient to find a subsystem *Sys* of S for which the TLA formula $\widehat{Sys} \Rightarrow P$ can be proven. This supports the broad applicability of the theorems supplied by the transfer protocol framework. Theorems have the form of TLA implications from a parameter condition $Pars$, a protocol mechanism subsystem *Sys*, and an environment condition invariant *EnvCond* to a service property specification (e.g., $Pars \wedge Sys \wedge \Box EnvCond \Rightarrow LiveInNoAttr$, cf. Fig. 8 in Sec. 7). The subsystems *Sys* of the theorems are relatively small and really concentrate on these protocol mechanisms which are necessary for the implementation of certain service properties. Besides of few parameter and non-blocking conditions the theorems do not refer to the environment of *Sys*. Therefore, a relatively small number of theorems can cover the relevant relations between protocol mechanisms and service properties. In practical proofs that a specific protocol implements a service with special functional properties,

therefore there is a very high probability that one finds suitable instances of framework theorems and consequently does not need to perform original proofs.

4 Transfer Protocol Framework

The Transfer Protocol Framework consists of specification modules and of theorems. The specification modules are cTLA process type declarations which describe protocol mechanisms, constraints of a basic transfer medium, and service constraints. They are structured into three layers:

- Service-Constraints (SCs):
  A service constraint (e.g., an instance of the process type *Corruptions* in Fig. 1) models a single property of a communication service. It is instantiated from an SC module of the framework. Service specifications are combined from SC instances. The framework contains SCs specifying that transmission errors (i.e., corruptions, duplicates, reorderings, gaps, phantoms) do not occur, that the service capacity is limited, and that the service guarantees a live delivery of transmitted user or signalling data. By other SCs one can model aspects of connection handling, datagram transfer, and functional quality of service. A third group of SCs allows to model constraints regarding the coordination of different connections (e.g., limiting the number of connections currently active in a station).

- Abstract Protocol Mechanisms (APMs) and Abstract Medium Constraints (AMCs):
  An abstract protocol specification is modeled by a system composed of APMs and AMCs. The APMs and AMCs reflect the common scenario of protocol descriptions. Protocol entities cooperate and communicate by means of a basic medium. With respect to this, the different APMs model single mechanisms of protocol entities and the AMCs describe properties of the basic medium. The APMs specify abstractions of protocol mechanisms used in real protocols. They only model the essential functions of the protocol mechanisms and do not attach importance to details of efficient implementation. For instance, the APM modeling sequence numbering is not concerned with the reuse of numbers and therefore is based on an infinite range of numbers. Besides sequence numbering, the framework contains APMs modeling protocol mechanisms used by various transfer protocols (cf. [8, 35]). One can specify the storage of user data, cyclic redundancy checks, segmentation and re-assembly of user data, as well as the handling of re-ordered and defective data. Moreover, APMs are available modeling the management of feedback messages (i.e., acknowledgement of data, reject of defective data units, granting new transmission credits), flow control mechanisms, and the handling of messages to trigger a feedback from the receiver to the transmitter of user data. Another group of APMs supports the specification of connection handling, datagram transfer, functional quality of service management, and mechanisms to coordinate different connections. Since the basic medium is a service, the AMCs model service contraints similarly to the SCs.

- Finite Abstract Protocol Mechanisms (FAPMs) and Abstract Medium Constraints (AMCs):

A system of FAPMs and AMCs models a transfer protocol in a quite direct manner. Each protocol mechanism of the protocol is modeled by an FAPM instance. AMC instances specify the basic medium. A composition of the FAPM instances and AMC instances forms a structured formal protocol specification. In contrast to the APMs, the FAPMs use only finite variables. Thus, sequence numbers have to be reused which corresponds directly to the mechanism employed in sliding window protocols. Furthermore, connections have to be identified by a finite number of connection identifiers. Therefore, besides FAPMs similar to the APMs listed above, the framework contains FAPMs modeling the administration of sequence numbers and connection identifiers.

The theorems of the framework are logical implications between cTLA systems. Due to the three-layered structure of the specifications patterns two kinds of theorems are used:

- An SC theorem states that a system modeled by APMs and AMCs implies a service constraint, i.e., an SC. For some SCs of the framework more than one theorem are available. That reflects that a service constraint can be realized by different protocol mechanism combinations.

- The gap between an abstract protocol system modeled by APMs and AMCs and a direct protocol model consisting of FAPMs and AMCs is bridged by APM and AMC theorems. A theorem states that a system of FAPMs and AMCs implements a certain APM or AMC.

The structure of the framework is further refined by distinguishing safety and liveness properties. Thus, each of the three sorts of specification modules contains two groups of modules called safety resp. liveness process types. Likewise the theorems are classified into safety and liveness theorems according to the process at the right side of the implication. The theorems are logical implications as listed below:

- Safety theorem: $Pars \wedge Sys \Rightarrow (Safety)Proc$,

- Liveness theorem: $Pars \wedge Sys \wedge \Box EnvCond \Rightarrow (Liveness)Proc$.

The system definition Sys, the central part of the left side of an theorem implication, describes a system composed from process instances. The theorem expresses that this system implements the process $Proc$ listed on the right side of the implication. The process instances of Sys are parametrized instantiations of process types. Since process types must not be parametrized arbitrarily, a predicate logic formula $Pars$ defines a sufficient condition for correct and consistent process parametrizations. Furthermore, in liveness theorems the left side of the implication contains an invariant, the so-called environment condition $EnvCond$. This condition constrains the behaviour of the environment of the system Sys. It rules out that fair actions of Sys may be blocked by the environment too often.

Currently, the transfer protocol framework contains 133 specification patterns (28 SCs, 44 APMs, 14 AMCs, and 47 FAPMs) and 165 theorems (31 SC theorems and 134 APM theorems).

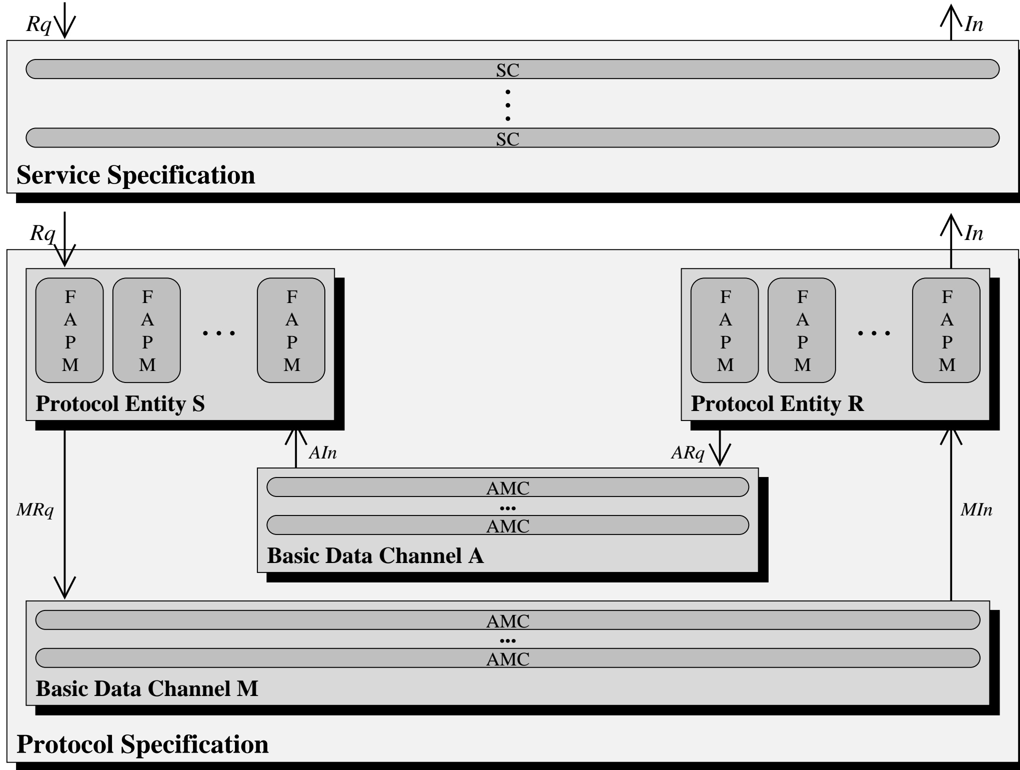

Figure 4: Structure of service and protocol specifications

5 Service Specification

Our protocol example shall implement a reliable and live simplex data transfer service of a fixed capacity. Particularly, data must be transmitted without errors. Thus, the service has to fulfill five service constraints: transmitted data are not corrupted; the stream of data delivered to the receiver does not contain gaps; the delivered data are not reordered; duplicates of data are not delivered; the stream of delivered data does not contain phantoms. Another service constraint limits the capacity of the service. At the same time only a certain number of data units — eight in this example — may be sent and not yet delivered. The last service constraint reflects liveness aspects. The service is alive if it eventually delivers all transmitted data units to the receiver guaranteeing that data may not be lost due to inactivity. Furthermore, due to the liveness of the service the transmitting service user will be able to submit new data units in intervalls since each data delivery causes the amount of data falling below the capacity limit. Thus, after a delivery at least one new data unit may be transmitted.

To create a formal specification of this service, at first we design constraint specifications, each modeling only one of the service constraints listed above (cf. upper part of Fig. 4). As already mentioned, the constraint specifications are developed parametrizing SCs of the transfer protocol framework. In a second step we compose the constraint specifications to the service specification.

The specification *SlidWindService*, listed in Fig. 3, models the service. It contains a parameter `usd` defining the set of data units to be transmitted by the service. For instance, the replacement of `usd` by the set $\{0, \ldots, 255\}$ specifies a byte oriented data transfer ser-

vice. The service specification *SlidWindService* consists of the eight constraints *Id*, *C*, *G*, *R*, *D*, *P*, *Cap*, and *LIn* which will be developed by instantiating the framework SCs *SDUId*, *Corruptions*, *Gaps*, *Reorderings*, *Duplicates*, *Phantoms*, *Capacity*, resp. *LiveInNoAttr*.

Id is a special constraint which models the assignment of an unambiguous and ordered sequence number to each data unit submitted. These sequence numbers are necessary to detect transmission errors. The other seven constraints processes specify the seven service constraints mentioned above. For example, the constraint *C* models that only data which are not corrupted during the transmission may be delivered to the service user. It is an instantiation of the SC *Corruptions*, listed in Fig. 1, which uses the process parameter `usd` and `tc`. `usd` corresponds to the global parameters `usd` in the process *SlidWindService*. By the relation `tc` we can fix, if and to which degree corruptions of data may be tolerated. For instance, if one models byte oriented data transfer, the replacement of `tc` by $\{(k, 2{\cdot}(k\ \mathbf{div}\ 2))\} \cup \{(k, 2{\cdot}(k\ \mathbf{div}\ 2{+}1))\}$ specifies that one tolerates the corruption of the lowest priority bit. Since in our example corruptions must not occur, we replace `tc` by the identity relation `{ (k,k) | k ∈ usd }`. The constraint process *G* models the service constraint that the stream of delivered data has no gaps. *G* is instantiated from the SC *Gaps*, the parameter *tg* of which describes the maximum size of gaps. We replace it by the value 0. The Constraints *R*, *D*, and *P* model the service constraints which exclude reoderings, duplicates, and phantoms in the stream of delivered data. The capacity of the service is modeled by the constraint *Cap* while the liveness constraint process *LIn* of type *LiveInNoAttr* (Fig. 2) specifies that the service must be alive.

The actions `Rq`, `fIn`, and `nIn` are specified in the service specification *SlidWindService*. By `Rq` the submission of a new data unit `d` by the transmitting service user is modeled. During the submission the sequence number `krq` is assigned to the data unit `d`. `Rq` is a conjunction of the process actions `Rq` of the process constraints *Id*, *C*, *Cap*, and *LIn* while the other processes do not contribute with a process action. cTLA, however, demands that the processes *G*, *R*, *D*, and *P* perform a local stuttering step during the execution of `Rq`. Thus, it models concurrency by interleaving.

The delivery of data units `d` with sequence numbers `krq` to the receiving service user is specified by the actions `fIn` and `nIn`. Since the service should be performed lively, we have to provide the action modeling the data delivery with a fairness assumption. As described in Sec. 2, we reduce this action to a weak fair action `fIn` and to a complementary non-fair action `nIn`. In both actions the constraint processes *Id*, *C*, *G*, *R*, *D*, *P*, and *Cap* participate with their process actions `In`. From the process *LIn* the process action `fIn` is coupled to the system action `fIn` and the process action `nIn` to `nIn`. `fIn` models the delivery of data essential for the progress of the transmission. For instance, the delivery of a data unit which was sent but not yet delivered is modeled by `fIn`. In contrast, `nIn` specifies the delivery of data not important for the progress of the transmission (fi., the delivery of duplicates). Since in our example all data deliveries are essential, the action `nIn` is never enabled. Thus, we can omit it in a simplified description.

The action `fIn` of system *SlidWindService* is weak fair since it contains the process action `fIn` of the process *LIn* which is weak fair as well (cf. Fig. 2). Therefore, if `fIn` is continuously enabled, it will eventually be selected for execution.

6 Protocol Specification

The sliding window protocol [42] shall implement the data transfer service introduced above. The protocol consists of a transmitter entity S at the station of the data transmitter and a receiver entity R at the receiver site which communicate by means of a full duplex basic service (cf. lower part of Fig. 4). The basic service is modeled by two simplex data channels M and A. It is less reliable than the service to be provided and we assume that it guarantees only that corruptions, reorderings, and phantoms do not occur during the transmission. Thus, data losses and duplicates are possible. We model the basic service composing the following service constraints: data units are not corrupted; data units are delivered in correct order; the stream of delivered data does not contain phantoms. This kind of data transfer quality is quite realistic. It exists if the point-to-point data transfer used is protected against corruptions but not against data losses due to buffer overflows in transit nodes. Duplicates may occur due to incorrect copying of data in the transit nodes. Furthermore, we demand that the basic service guarantees the following liveness assumption: If the user of the basic service repeatedly sends data units with a certain attribute, eventually one of these data units will be delivered to its peer. Without this liveness assumption, data units essential for the transmission might be lost again and again leading to livelocks.

The main task of the protocol entities S and R of the sliding window protocol is to detect and to remedy data losses and duplicates. Furthermore, the capacity and liveness of the service provided by the protocol have to be guaranteed. The protocol uses a set of protocol functions customary for most modern data transfer protocols (cf. [8, 35]). Data losses and duplicates are detected by means of sequence numbers. Each new data unit submitted to the protocol entity S is assigned a sequence number. The data unit and its sequence number are transmitted together in a protocol data packet, a so-called protocol data unit (PDU). Due to the order of the sequence numbers the receiver entity R detects duplications and gaps. Duplicated data will be ignored while gaps are remedied by selective retransmission. The protocol applies the method "Positive Acknowledgement with Retransmission" (PAR). In intervalls S retransmits data units which are still not confirmed. To confirm data units, R transmits confirmation PDUs to S which contain the sequence number `sack` of the data unit delivered last to the service user. By this PDU all data units are confirmed, of which the sequence numbers are lower or equal to `sack`. S guarantees the finite capacity of the service provided since it does only accept a new data unit for transmission if less than eight data units are currently not confirmed. The liveness of the service is guaranteed since on the one hand S transmits new data units and retransmits unconfirmed data units to R in intervalls. On the other hand R delivers all correct data units to the service user and repeatedly transmits confirmation PDUs to S.

The specification *SlidWindProtocol* of the sliding window protocol (Fig. 5 and Fig. 6) is designed by parametrizing and composing processes of the framework[1]. Like the service specification, *SlidWindProtocol* contains a parameter `usd` denoting the set of data units transfered by the protocol. The parameter replacements of the process instances

[1]Due to the size of the specification we abstain from listing most process parameters in the `PROCESSES` part. In the `ACTIONS` part most system actions are listed without action parameters and local process actions.

```
! Specification of the Sliding-Window-Protocol
PROCESS SlidWindProtocol (usd : any)  ! usd : set of user data transfered
  IMPORT SWParameters(usd);

  PROCESSES
! FAPMs : Modeling transmitter entity S
    SBK : SBufferKey    ! Sequence number handler of the transmitter entity
            (swpdu, swpci, usd, swpdu, swspci, swskey, 1, swskk, swskn,
             swskm, swusdsize, 1, 16, 8);
    SBU : SBufferUsd(...); ! User data storage handler of the transmitter entity
    SAck : SAcknowledge(...); ! Data acknowledge mechanism of the transm. entity
    SCap : SCapacity(...); ! Preventing data unit overflow in the transm. entity
    SLMRq : SLiveMRq(...); ! Liveness of the transmitter entity guaranteeing the
                           ! transmission and retransmission of data
! FAPMs : Modeling receiver entity R
    RBK : RBufferKey(...); ! Sequence number handler of the receiver entity
    RBU : RBufferUsd(...); ! User data storage handler of the receiver entity
    RG : RGaps(...); ! No gaps of delivered data
    RR : RReorderings(...); ! Delivered data is not reordered
    RD : RDuplicates(...); ! No duplications in delivered data
    RP : RPhantoms(...); ! no phantoms delivered by the receiver entity
    RAck : RAcknowledge(...); ! Data acknowledge mechanism of the rec. entity
    RLARq : RLiveARq(...); ! Liveness of the receiver entity guaranteeing the
                           ! transmission of acknowledgement data
    RLIn : RLiveIn(...); ! Liveness of the receiver entity guaranteeing the
                         ! delivery of received data to the service user

! AMCs : Constraints of the basic service channel M
    MS : MSDUId; ! Ordered assignment of sequence numbers
    MC : MCorruptions(...); ! No corruptions during data transfer on channel M
    MR : MReorderings(...); ! No reorderings during data transfer on channel M
    MP : MPhantoms(...); ! No phantoms generated on channel M
    MLI : MLiveIn(...); ! Liveness of channel M guaranteeing that each pdu
                        ! sent in intervalls will be eventually delivered
! AMCs : Constraints of the basic service channel A
    AS : ASDUId; ! Ordered assignment of sequence numbers
    AC : ACorruptions(...); ! No corruptions during data transfer on channel A
    AR : AReorderings(...); ! No reorderings during data transfer on channel A
    AP : APhantoms(...); ! No phantoms generated on channel A
    ALI : ALiveIn(...); ! Liveness of channel A guaranteeing that each pdu
                        ! sent in intervalls will be eventually delivered

  ACTIONS
    ...;
END
```

Figure 5: Protocol Specification *SlidWindProtocol* (PROCESSES-part)

```
PROCESS SlidWindProtocol (usd : any)  ! usd : set of user data transfered
  IMPORT SWParameters(usd);

  PROCESSES
    ...;

  ACTIONS
    Rq (krq : fkey; d : usd) ≜
             ! Submission of user data d with sequence number krq
      SBK.Rq (krq,d) ∧ SBU.Rq (krq,d) ∧ SAck.Rq (krq) ∧ SCap.Rq (krq) ∧
      SLMRq.Rq (krq,d) ∧ ...;
    fIn (krq : fkey; d : usd) ≜ ...;    nIn (krq : fkey; d : usd) ≜ ...;
             ! Delivery of user data d with sequence number krq to the
             ! service user (Weak fairness in fIn)
    fMRq ( ... ) ≜ ...;    nMRq ( ... ) ≜ ...;
             ! Submission of a pdu to channel M (Strong fairness in fMRq)
    fMIn ( ... ) ≜ ...;    nMIn ( ... ) ≜ ...;
             ! Delivery of a pdu including data units sent via channel M to the
             ! receiver entity (Weak fairness in fMIn)
    fARq ( ... ) ≜ ...;    nARq ( ... ) ≜ ...;
             ! Submission of a pdu including acknowledgements of data
             ! units to channel A. (Strong fairness in fARq)
    fAIn ( ... ) ≜ ...;    nAIn ( ... ) ≜ ...;
             ! Delivery of a pdu including acknowledgements of data units
             ! sent via channel A (Weak fairness in fAIn)
    MTick ≜ MLI.MTick ∧ ...;    ATick ≜ ALI.ATick ∧ ...;
             ! Internal actions indicating the loss of a pdu in M resp. A
    fMNoTick (p : [info : usd; seq : key ∪ {"≪≫"}; ack : key]) ≜ ...;
    fANoTick (p : [info : usd; seq : key ∪ {"≪≫"}; ack : key]) ≜ ...;
             ! Internal actions of M resp. A guaranteeing the delivery of p if
             ! pdus containing the same data unit are sent often. (Strong F.)
END
```

Figure 6: Protocol Specification *SlidWindProtocol* (ACTIONS-part)

are defined in a separate process *SWParameters* listed below. In the `PROCESSES` part of the specification (cf. Fig. 5) the processes modeling the entities and the basic channels are listed. The transmitter protocol entity S is specified by the processes *SBK*, *SBU*, *SAck*, *SCap*, and *SLMRq* instantiated from FAPMs of the framework. They model protocol mechanisms to maintain sequence numbers as well as data units, to handle confirmation PDUs, to limit the size of the transmitter buffer, and to guarantee the liveness by transmitting data units in intervalls.

The receiver protocol entity is modeled by the processes *RBK*, *RBU*, *RG*, *RR*, *RD*, *RP*, *RAck*, *RLARq*, and *RLIn* which are also instantiated from FAPMs. These processes specify the protocol mechanisms to maintain sequence numbers as well as data units, to deliver data to the service users in the correct order, and to confirm delivered data.

```
! Parameters of the Specification of the Sliding-Window-Protocol
CONSTANT MODULE SWParameters (usd : any)  ! usd : set of user data transfered

  CONSTANTS
! Used in FAPMs and AMCs
    swpdu ≜ [info : usd; seq : fkey ∪ {"≪≫"}; ack : fkey];
      ! Abstract PDU format
    swpci ≜ [seq : fkey ∪ {"≪≫"}; ack : fkey];
      ! Protocol Control Information (PCI)
    swspci ≜ [ x ∈ [info : usd; seq : fkey ∪ {"≪≫"}; ack : fkey]
                ↦ [ seq ↦ x.seq; ack ↦ x.ack ] ];
      ! Pointer to the PCI of a PDU
    ...;
! Used in AMCs
    swtc ≜ { (k,k) | k ∈ [info : usd; seq : key ∪ {"≪≫"}; ack : key] };
      ! Relation: identy of PDUs
    ...;
END
```

Figure 7: Parameter definitions *SWParameters*

Furthermore, *RLARq* and *RLIn* guarantee the liveness of the entity since confirmation PDUs and data deliveries are repeatedly triggered.

The basic data channel M models the transfer of user data and so-called protocol control information (PCI) within PDUs from S to R. We specify it by means of the processes *MId*, *MC*, *MR*, *MP*, and *MLI* which are instantiated from AMCs of the framework. Like the SC *Id*, *MId* is a special constraint modeling the assignment of data units to PDUs to be transmitted. By *MC*, *MR*, and *MP* we model that transmitted data units are neither corrupted nor reordered and that phantoms are not delivered. *MLI* guarantees that, if PDUs with a certain attribute are sent in intervalls, eventually one of them will be delivered to the protocol entity R. The basic data channel A modeling the transfer of PCI within PDUs from R to S is specified by the processes *AId*, *AC*, *AR*, *AP*, and *ALI* which correspond to the constraints of channel M.

The coupling of process actions to system actions is described in the part of the specification headed by `ACTIONS` (cf. Fig. 6). The actions `In`, `MRq`, `MIn`, `ARq`, and `AIn` are each specified by a fair and a complementary non-fair action. Thus, `fMRq` specifies the transmission of PDUs which are essential for the progress of the communication. In contrast, `nMRq` models transmissions which are allowed and for the sake of efficiency often are also desirable but at the present state not important for the liveness of the protocol. The specification respects the distribution of the protocol entities S and R since either the processes of S or those of R participate in a system action by stuttering steps only.

The constraints are developed from framework processes by instantiation of process parameters. In Fig. 7 the specification *SWParameters* lists the definitions of the actual parameter types. The identifier `swpdu` models the format of the PDUs of the sliding window protocol. It is a record consisting of the three components `info`, `seq`, and `ack`.

Data units transmitted in the PDU are stored in the record component info. In the element seq either the sequence number of a data unit is stored which can be a natural number or the special symbol "notsent" denoting a phantom (data type key). Or the special symbol "<<>>" marks that the PDU does not contain a data unit. The sequence number of the data unit delivered last to the service user is stored in the record component ack.

The sequence numbers of data units sent resp. confirmed form the protocol control information (PCI) of a PDU. Thus the identifier spci defines a record consisting only of the components seq and ack. swspci models a pointer to the PCI of a PDU. It maps a PDU record to the PCI record, of which the components seq and ack contain the same values as in the PDU.

As already mentioned in Sec. 4, we specify the sliding window protocol in two steps. First, we create the detailed protocol specification. In a second step we develop a more abstract protocol specification which models the distributed functionality of the entities but contains variables of an infinite range. Thus, we abstract from protocol errors due to the reuse of sequence numbers and connection identifiers in the abstract specification. The abstract specification supports the reduction of the protocol verification into two simpler steps (cf. Sec. 7). In the abstract protocol specification we model the protocol entities by the constraint processes *SBK*, *RBK*, *SBU*, *RBU*, *RG*, *RR*, *RD*, *SAck*, *RAck*, *SCap*, *SLMRq*, *RLARq*, and *RLIn* again which, however, are instantiated from APMs this time. To model the basic data channels M and A, we use the same AMCs as in the specification *SlidWindProtocol*.

7 Verification

The protocol verification guarantees that the communication service specified in Sec. 5 is implemented by the sliding window protocol described in Sec. 6. Due to the compositionality of cTLA we can reduce the verification into a series of simpler proof steps. In each proof step we verify that a single service constraint is realized by a protocol subsystem consisting of only some protocol mechanisms. Each proof step corresponds directly to a framework theorem. We assume that the theorems are correct though we have to refer to the reservations concerning the stringency of the presently available theorem proofs described in the introduction. Therefore, we have to check only if the protocol specification contains all protocol mechanisms necessary for the service constraint. Furthermore, the actual parameters of the protocol mechanisms have to be consistent to each other and to those of the service constraint realized. Thus, we can reduce the protocol verification into the simple selection and consistency checking of framework theorems.

The protocol verification is reduced to two major steps. First, we prove that the communication service is fulfilled by the abstract protocol modeled by APMs and AMCs. Second, we verify that the abstract protocol specification is implemented by the more detailed sliding window protocol specification consisting of FAPMs and AMCs. To perform the first step, we apply eight framework theorems, each proving one SC of the service specification. As an example we list the theorem instance verifying an instance of the

liveness SC *LiveInNoAttr*[2] in Fig. 8. The theorem states that an instance of the SC *LiveInNoAttr*, i.e., *LIn*, is implemented by a protocol system which contains the processes of *Sys* as a subsystem if the conditions *Pars* and $\Box$*EnvCond* hold. *Sys* consists of instances of the APMs *SLiveMRq*, *RLiveARq*, *RLiveIn*, and *RAcknowledge* as well as instances of the AMCs *MSDUId*, *MCorruptions*, *MPhantoms*, *MLiveIn*, *ASDUId*, *ACorruptions*, *APhantoms*, and *ALiveIn*. It guarantees that user data (*SLiveMRq*) and confirmations (*RLiveARq*) are transmitted arbitrarily often between the protocol entities, that correctly transmitted user data are delivered to the service user (*RLiveIn*), and that only delivered data are confirmed (*RAcknowledge*). Furthermore the basic data channels are alive (*MLiveIn* and *ALiveIn*) and do not deliver corrupted data or phantoms (*MSDUId*, *MCorruptions*, *MPhantoms*, *ASDUId*, *ACorruptions*, and *APhantoms*).

While instantiating the formal parameters of the constraint processes in *Sys* according

[2] To reduce the specification size, we omitted the actual parameters of the processes in the protocol subsystem definition.

```
LET Pars ≜ {(p,q)| p, q ∈ [info : usd; seq : key ∪ {"≪≫"}; ack : key] ∧
                   p.seq = q.seq } =
           {(p,q)| p.seq = q.seq ∧
                   p ∈ [info : usd; seq : key ∪ {"≪≫"}; ack : key] ∧
                   q ∈ [info : usd; seq : key ∪ {"≪≫"}; ack : key] }
           ∧
           { (k,k) | k ∈ [info : usd; seq : key ∪ {"≪≫"}; ack : key] } ⊆
           { (p,q) | q ∉ [info : usd; seq : key ∪ {"≪≫"}; ack : key]  ∨
                     p.seq = q.seq }
           ∧
           { (k,k) | k ∈ [info : usd; seq : key ∪ {"≪≫"}; ack : key] } ⊆
           { (p,q)| q ∉ [info : usd; seq : key ∪ {"≪≫"}; ack : key]  ∨
                    p.ack = q.ack };
    Sys ≜  SLiveMRq ([info : usd; seq : key ∪ {"≪≫"}; ack : key], ...) ∧
           RLiveARq (...) ∧ RLiveIn (...) ∧ RAcknowledge (...) ∧
           MSDUId ∧ MCorruptions (...) ∧ MPhantoms (...) ∧ MLiveIn (...) ∧
           ASDUId ∧ ACorruptions (...) ∧ APhantoms (...) ∧ ALiveIn (...) ∧
           CC_LiveInNoAttr;
    EnvCond ≜
           ∀ krq,p,kd : Enabled(SLiveMRq.fMRq(krq,p,kd)) ⇒
              (krq,p,kd) ∈ Sys.e_fMRq ∧
           ∀ p,kd : Enabled(RLiveARq.fARq(p,kd)) ⇒ (p,kd) ∈ Sys.e_fARq ∧
           ∀ krq,d : Enabled(RLiveIn.fIn(krq,d)) ⇒ (krq,d) ∈ Sys.e_fIn ∧
           ∀ krq : Enabled(MLiveIn.fMIn(krq)) ⇒ krq ∈ Sys.e_fMIn ∧
           ∀ d : Enabled(MLiveIn.fMnoTick(d)) ⇒ d ∈ Sys.e_fMnoTick ∧
           ∀ krq : Enabled(ALiveIn.fAIn(krq)) ⇒ krq ∈ Sys.e_fAIn ∧
           ∀ d : Enabled(ALiveIn.fANoTick(d)) ⇒ d ∈ Sys.e_fANoTick;
IN Pars ∧ Sys ∧ □ EnvCond ⇒ LiveInNoAttr
```

Figure 8: Theorem *LiveInNoAttr*

to the description in Sec. 6, we adapted the theorem to our example. Since the abstract protocol specification contains instances of all APMs and AMCs listed in *Sys* which are coupled in accordance with the (not explicitely listed) coupling formula $CC_{LiveInNoAttr}$, *Sys* is a subsystem of the abstract sliding window protocol. The replacements of the parameters are consistent if the formula *Pars* holds. The first conjunct of *Pars* is a tautology and therefore holds. The other conjuncts express that, if two PDUs p and q are identical, their record components $p.seq$ and $q.seq$ resp. $p.ack$ and $q.ack$ must have equal values as well. These conjuncts hold trivially since two records are equal per definitionem if all record components (particularly *seq* and *ack*) are equal as well.

The temporal condition $\Box EnvCond$ guarantees that *LiveInNoAttr* is implemented not only by the subsystem *Sys* but also by the whole abstract sliding window protocol. A liveness proof might fail if the entire system contains processes weakening the liveness of the APMs and AMCs in *Sys*. For instance, if the entire system contains an APM preventing the delivery of user data at all, *Sys* could of course not guarantee the liveness of the SC *LiveInNoAttr*. During the design of the theorem we verified that except for the APM *DataChanOpenR* all APMs and AMCs of the framework fulfill the condition $\Box EnvCond$. *DataChanOpenR*, however, is not a part of the abstract protocol specification. Thus, $\Box EnvCond$ also holds, and we proved that the abstract sliding window protocol implements the service constraint *LIn* modeled as an instance of the SC *LiveInNoAttr*. In the same way we verify the other seven SCs of the service specification.

Thereafter, the second major proof step verifies that the abstract sliding window protocol specification is fulfilled by the more detailed one. It is performed accordingly. By application of 13 framework theorems we prove that the 13 APMs of the abstract protocol specification are fulfilled. The proof of the AMCs is not necessary since the basic services used by the entities of the detailed resp. abstract protocol are modeled by identical AMCs.

8 Conclusion

We outlined the essential features of the transfer protocol framework and its application to the formal specification and verification of communication protocols with the help of a sliding window protocol example. Similarly, more complex protocols were examined with remarkable few expense of work. For instance, the high-speed transfer protocol XTP [44] was specified and verified within three weeks [24]. The framework can be accessed via WWW (http://ls4-www.informatik.uni-dortmund.de/RVS/P-TPM).

Currently, we extend the specification technique cTLA. Besides the modeling of event-discrete, not time-valued dynamical behaviours, cTLA can also be used to specify real-time properties and continuous behaviours [19, 23]. We are going to adapt the framework approach to the modeling of distributed realtime systems. At the moment we examine the application field of chemical engineering system control [25]. Furthermore, the cTLA extension can also be utilized for communication protocols (fi., modeling the transmission of multimedia data).

References

[1] B. Alpern, F.B. Schneider, Defining liveness, Inform. Proc. Letters 21(1985) 181-185.

[2] R.J.R. Back, R. Kurki-Suonio, Decentralization of process nets with a centralized control, Distrib. Comp. 3(1989) 73-87.

[3] M. Broy, F. Huber, B. Paech, B. Rumpe, K. Spies, Software and System Modeling Based on a Unified Formal Semantics, in: M. Broy, B. Rumpe, (Eds.), Proc. RTSE´97, LNCS 1526, Springer-Verlag, 1998.

[4] S. Budkowski, Estelle Development Toolset, Comp. Netw. and ISDN Sys. 25(1992) 63-82.

[5] K.M. Chandy, J. Misra, Parallel Program Design — A Foundation, Addison Wesley, 1988.

[6] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transac. on Prog. Lang. and Sys. 8(1986) 244-263.

[7] E.M. Clarke, O. Grumberg, and D.E. Long, Model checking and abstraction, ACM Transac. on Prog. Lang. and Sys. 16(1994) 1512-1542.

[8] W.A. Doeringer, D. Dykeman, M. Kaiserswerth, W. Meister, H. Rudin, R. Williamson, A Survey of Light-Weight Transport Protocols for High-Speed Networks, IEEE Transac. on Comm. 38(1990) 2025-2039.

[9] U. Engberg, P. Grønning, L. Lamport, Mechanical verification of concurrent systems with TLA, in: G. von Bochmann, D.K. Probst, (Eds.), Proc. CAV'92, Lecture Notes in Computer Science, vol. 663, Springer, Berlin, 1992 pp. 44-55.

[10] J.-C. Fernandez, L. Mounier, "On the fly" verification of behavioural equivalences and preorders, in: LNCS 575, Springer-Verlag, 1991 pp. 181-191.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns — Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[12] H. Garavel, M. Jorgensen, R. Mateescu, Ch. Recheur, M. Sighireanu, B. Vivien, CADP'97 — Status, Applications and Perspectives, in: COST247 Workshop on Appl. Form. Methods in Sys. Design, 1997.

[13] S.J. Garland, J.V. Guttag, J.J. Horning, An overview of Larch, in: P. Lauer, (Ed.), Functional Programming, Concurrency, Simulation, and Automated Reasoning, LNCS 693, Springer-Verlag, 1993 pp. 329-348.

[14] S.J. Garland, N.A.Lynch, M. Vaziri, IOA: a language for specifying, programming, and validating distributed systems, MIT Lab. for Comp. Sci., 1997.

[15] B. Geppert, R. Gotzhein, F. Rößler, Configuring Communication Protocols Using SDL Patterns, in: A. Cavalli, A. Sarma, (Eds.), Proc. SDL'97, Elsevier, 1997.

[16] R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, in: P. Dembiński, M. Średniawa, (Eds.), Proc. PSTV XV, Chapman & Hall, London, 1995 pp. 3-18.

[17] P. Herrmann, Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen, Deutscher Universitätsverlag, 1998 (in German).

[18] P. Herrmann, O. Drögehorn, W. Geisselhardt, H. Krumm, Tool-supported formal verification of highspeed transfer protocol designs, in: Proc. ICOTS'99, ATSMA, 1999 pp. 531-541.

[19] P. Herrmann, G. Graw, H. Krumm, Compositional Specification and Structured Verification of Hybrid Systems in cTLA, in: Proc. ISORC'98, IEEE Computer Society Press, 1998 pp. 335-340.

[20] P. Herrmann, T. Kraatz, H. Krumm, M. Stange, Automated Verification of Refinements of Concurrent and Distributed Systems, Research Report 541, Universität Dortmund, Fachbereich Informatik, 1994.

[21] P. Herrmann, H. Krumm, Compositional Specification and Verification of High-Speed Transfer Protocols, in: S.T. Vuong, S.T. Chanson, (Eds.), Proc. PSTV XIV, Chapman & Hall, London, 1994 pp. 339-346.

[22] P. Herrmann, H. Krumm, Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations, in: P. Dembiński, M. Średniawa, (Eds.), Proc. PSTV XV, Chapman & Hall, London, 1995 pp. 171-186.

[23] P. Herrmann, H. Krumm, Specification of Hybrid Systems in cTLA+, in: Proc. WPDRTS'97, IEEE Computer Society Press, 1997 pp. 212-216.

[24] P. Herrmann, H. Krumm, Modular Specification and Verification of XTP, Telecom. Sys. 9(1998) 207-221.

[25] P. Herrmann, H. Krumm, Formal Hazard Analysis of Hybrid Systems in cTLA, in: Proc. SRDS'99, IEEE Computer Society Press, 1999 pp. 68-77.

[26] C. Heyl, A. Mester, H. Krumm, cTc — A Tool Supporting the Construction of cTLA-Specifications, in: T. Margaria, B. Steffen, (Eds.), Proc. TACAS'96, LNCS 1055, Springer-Verlag, 1996 pp. 407-411.

[27] V. Hinz, Formale Spezifikation und Verifikation eines Hochleistungstransferprotokolls mit dem Transferprotokollframework, Diploma Thesis, University of Dortmund, 1998 (in German).

[28] G.J. Holzmann, Algorithms for Automated Protocol Verification, AT&T Tech. Journ. (1990) 32-44.

[29] G.J. Holzmann, The model checker Spin, IEEE Transac. on Soft. Eng. 23(1997) 279-295.

[30] ISO, Information technology — Open Systems Interconnection — ESTELLE: A formal description technique based on an extended state transition model (Amendment 1), International Standard ISO/IEC 9074, 1997.

[31] ISO, Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour, International Standard ISO 8807, 1989.

[32] ITU, SG X: Recommendation Z.100: CCITT Specification and Description Language SDL, 1993.

[33] R.P. Kurshan, Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach, Princeton University Press, 1995.

[34] L. Lamport, The Temporal Logic of Actions, ACM Transac. on Prog. Lang. and Sys. 16(1994) 872-923.

[35] T.F. La Porta, M. Schwartz, Architectures, features, and implementation of high-speed transport protocols, IEEE Netw. Mag. (1991) 14-22.

[36] T.F. La Porta, M. Schwartz, The MultiStream Protocol: A Highly Flexible High-Speed Transport Protocol, IEEE Journ. on Selec. Areas in Comm. 11(1993).

[37] A. Mester, H. Krumm, Composition and Refinement Mapping based Construction of Distributed Applications, in: U.H. Engberg, K.G. Larsen, A. Skou, (Eds.), Proc. TACAS'95, BRICS Notes Series, vol. NS-95-2, 1995 pp. 290-303.

[38] A. Mester, H. Krumm, Formal behavioural patterns for the tool-assisted design of distributed applications, in: H. König, K. Geihs, T. Preuß (Eds.), Proc. DAIS 97, Chapman & Hall, London, 1997 pp. 235-48

[39] S. O´Malley, L. Peterson, A Dynamic Network Architecture, ACM Transac. on Comp. Sys. 10(1992) 110-143.

[40] S. Owre, J. Rushby, N. Shankar, F. von Henke, Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS, IEEE Transac. on Soft. Eng. 21(1995) 107-125.

[41] T. Plagemann, A Framework for Dynamic Protocol Configuration, to appear in Eur. Trans. on Telecomm. (1999).

[42] A. S. Tanenbaum, Computer Networks, Prentice-Hall, 1996.

[43] C.A. Vissers, G. Scollo, M. van Sinderen, Architecture and specification style in formal descriptions of distributed systems, in: S. Agarwal, K. Sabnani, (Eds.), Proc. PSTV VIII, Elsevier, Amsterdam, 1988 pp. 189-204.

[44] XTP Transport Protocol Specification, Revision 4.0, XTP Forum, Santa Barbara, 1995.

[45] M. Zitterbart, B. Stiller, A. Tantawy, A Model for Flexible High-Performance Communication Subsystems, IEEE Journ. on Selec. Areas in Comm. 11(1993) 507-518.