In Proceedings of the *7th International Conference on Telecommunication Systems — Modelling and Analysis*, pages 531-541, ATSMA, IFIP, 1999

Tool-supported formal verification of highspeed transfer protocol designs

Peter Herrmann[†], Olaf Drögehorn[*], Walter Geisselhardt[*], Heiko Krumm[†]

[†] Universität Dortmund, Email: {herrmann|krumm}@ls4.cs.uni-dortmund.de
[*] Universität–GH Duisburg, Email: {droege|gd}@uni-duisburg.de

Abstract

The so-called transfer protocol framework facilitates the development of formal highspeed communication protocol and service descriptions. It contains specification modules which can be easily combined to specifications of protocols or services. Additionally, the framework supports the formal protocol verification. The proof, that a protocol specification implements a service specification, can be reduced into a set of lemmata which correspond directly to theorems offered by the framework, too.

This contribution centers on the application of the framework to perform protocol verifications. We will introduce the tool COAST which simplifies verifications by selecting and checking mechanically suitable framework theorems. To clarify the application of COAST, we will outline the verification of the complex real-life high-speed data transfer protocol XTP.

1 Introduction

Due to the high performance demands of modern highspeed and multimedia applications many new data transfer protocols and protocol variants were recently developed. Since most of these protocols are very complex, one should support their design by formal methods (cf. [15]). In reality, however, protocols are frequently developed without any formal support, although standardized formal description techniques (i.e., ISO/OSI: Estelle [27] and Lotos [28], ITU: SDL [29]) are available. Therefore protocols are often designed by means of incomplete and ambiguous protocol descriptions. Furthermore, protocol developers often omit the development of abstract service specifications, which describe the communication services to be provided by the protocols. If, however, a service specification is lacking, there is no way to check the correctness of the protocol with respect to the services to be provided. Thus, design errors are often detected in later development phases only leading to costly delays.

In practice, the application of formal techniques is mostly omitted, since the application of formal techniques introduces additional efforts. While some formal techniques are supported by tools (cf. [2, 3, 10, 12, 16, 26]), either the tools cannot handle the complexity of a modern highspeed protocol, or subtle formal models and descriptions of protocols have to be developed in a creative manner, which is quite expensive. The transfer protocol framework [17, 21] facilitates the development of formal specifications. According to program development support by libraries of reusable program modules, a communication protocol or service is specified by instantiating and combining suitable specification modules, which are contained in the framework. Protocol specifications consist of modules each modelling a single protocol mechanism (e.g., sequence numbering of protocol data units, repeat requests, credit granting; cf. [7]). Service specifications are combined from modules which describe single service constraints (e.g., no corruptions of transmitted data, correct order of delivered data, liveness of the service). Thus, the protocol developer does not need to create a protocol or service specification from scratch. Instead, one uses modules of the framework and concentrates only on suitable parameter instantiations and on the combination of different modules. By this method, the logical structure of a protocol or service is modelled quite directly. Thus, the framework supports the understanding of the cooperation of the various system modules.

The development of separate service and protocol specifications also facilitates the protocol verification which proves that a protocol actually provides a service. The protocol verification may be performed either mechanically based on reachability analysis (e.g., state space exploration [25], model checking [5, 6, 11, 14, 19]) or by symbolical logical reasoning. Since most practically relevant highspeed protocols exceed the limitations of automated tools, these protocols can only be verified logically affording a high work-load. The verification, however, can be significantly facilitated by the framework taking into consideration that both the service and the protocol specifications are combined from existing framework modules. The framework contains theorems each stating that a service constraint is fulfilled by a protocol subsystem which is combined from certain protocol mechanisms. Since for each possible pair of a service constraint and a protocol subsystem the framework contains an already proven theorem, one can accomplish a protocol verification by identifying a suitable protocol subsystem as well as a theorem for each constraint of the service specification. By this method even complex highspeed transfer protocols like XTP can be verified easily [23]. Moreover, this verification method supports the understanding of a protocol, since the logical relations between service constraints and protocol subsystems are emphasized.

After selecting suitable theorems for the verification that a protocol specification fulfills a service specification, the protocol developer performs two or three checks for each theorem. First, one examines if the protocol subsystem quoted in a theorem is a subsystem of the protocol specification. In particular, one checks that each of the specification modules, which form the protocol subsystem, is also contained in the protocol specification. The second check corresponds to the proof of a condition which assures that specification modules of the service constraint and the protocol mechanisms are instantiated consistently. The third check is only performed if a theorem is used to prove service constraints which state liveness properties (cf. [1]). By this check one guarantees that the protocol specification does not contain specification modules spoiling the liveness property. In all protocol verifications, performed up to now, these checks were very easy.

Since in a protocol proof usually a large number of theorems has to be arranged (fi. 85 in the verification of XTP), tool-support for the selection and checking of the theorems is of interest. The tool COAST (Consistency of a specification in cTLA+) [8] performs this task. The input files of COAST are a service and a protocol specification both modelling an instantiation and a combination of specification modules of the framework. According to the input specifications, COAST selects suitable theorems from a database of all framework theorems and performs mechanically the first and (in proofs of liveness properties) the third check. The second check, however, cannot be automized since a logical formula has to be proven. Therefore COAST translates the formula into the syntax of a frontend tool [13] for the theorem prover OTTER [35]. OTTER tries to verify the formula by deduction which, due to the simplicity of this proof, can be usually performed without any interactive user support. If COAST can select and check (with the aid of OTTER) a framework theorem for each module of the service specification, the protocol verification succeeds.

The transfer protocol framework approach applies the modular specification technique cTLA (compositional TLA) [20, 36] which is based on Leslie Lamport's TLA (Temporal Logic of Actions) [33]. The specification modules of the framework are cTLA modules which contain generic parameters. Under the instantiation of these parameters the cTLA modules model process instances which are composed to service or protocol specifications. Similarly to LOTOS [28] and to [30], the processes interact via synchronous joint actions.

In the sequel we outline cTLA and the framework concisely. Thereafter, we describe the functionality and the structure of COAST. Moreover, the application of COAST will be clarified by outlining the verification of the protocol example XTP.

2 cTLA

In cTLA, specification modules describe processes. A process is modelled by a state transition system. As an example we refer to the definition of the process C in Fig. 1. This process specifies the service constraint that, except for phantoms, all delivered data units are transmitted between the users of a service without corruptions.

The syntax of cTLA is oriented at programming languages like Modula 2. The process header consists of the keyword PROCESS, the process name C, and optionally the list of generic parameters. In our example the generic parameter usd models the set of data units which can be transfered

```
PROCESS C ( usd : Any ) ! usd : set of user data transfered
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 sequence no. krq
      buf′ = buf ∪ {(krq,d)} ;
    In (krq : key; d : usd) ≜
    ! Delivery of user data d with sequence number krq
      ( krq = "notsent" ∨ ∀ e ∈ usd :: ((krq,e) ∉ buf) ∨
        (krq,d) ∈ buf ) ∧
      buf′ = buf ;
END
```

Figure 1: Safety process C

between two service users. The keyword `IMPORT` refers to the inclusion of other modules (i.e., *Symbols*) which contain definitions of data types, functions, and constants. The state space of a process is modelled by variables which are declared in the section `VARIABLES`. In our example process C `buf`, which describes a set of pairs of a sequence number[1] and an user data unit, is the only variable. In `buf` all data units are stored which were ever sent by the transmitting service user. A predicate headed by the construct `INIT` models the set of initial states. In C, only the state, where `buf` corresponds to the empty set, is initial. The state transitions are modelled by actions which are defined in the section `ACTIONS`. An action models a set of transitions. It is a predicate about a pair of an actual state and a next state. The actual state is referenced by variables (fi. `buf`). The next state is referenced by so-called primed variables (fi. `buf'`). A pair of an actual and a next state, the variables of which fulfill the predicate, is a state transition of the action. Action definitions can contain data parameters. In our example the action `Rq` corresponds to the submission of data units. For instance, `Rq(2,"data")` describes the submission of the data unit `"data"` and the assignment of the sequence number `2` to this data unit. The variable `buf` in the next state contains the pairs of `buf` in the actual state and additionally the pair `(2,"data")`. The action `In` models the delivery of a data unit `d` with the sequence number `krq`. A data unit may be delivered only if it is either a phantom message (`krq = "notsent"` ∨ ∀ `e` ∈ `usd` :: ((`krq`,`e`) ∉ `buf`)) or if the delivered data is not corrupted during the transmission ((`krq`,`d`) ∈ `buf`). In addition to the actions defined in the section `ACTIONS`, a cTLA process may also perform the so-called stuttering step, the execution of which does not change the process state.

With respect to separate safety and liveness properties (cf. [1]), the process C models only safety properties. Thus, C tolerates state sequences, where after a finite number of state changes only stuttering steps are performed (i.e., the process is suddenly terminated). To rule out such state sequences, process actions can be attributed with fairness assumptions. After the action definition section of a process, one adds description constructs of the form `WF: In;` or `SF: In;`. The WF construct expresses that an action (i.e., `In`) has to be executed weak fairly. A weak fair action must be performed eventually if it would otherwise be continuously be enabled for an infinite period of time. By the SF construct an action is declared to be strong fair. It has even to be performed, if it is disabled from time to time. Weak fair and strong fair actions were introduced in [1]. In contrast to direct liveness properties, these fairness assumptions guarantee not to be contradictory to the safety properties of a process. Therefore, in TLA [33] and cTLA liveness properties are generally modelled by weak and strong fair actions. Moreover, the processes of the transfer protocol framework either model safety properties or liveness properties only.

In cTLA, processes are combined to systems similarly to Lotos [28]. The processes interact via synchronous joint actions. The transfer of data between processes is modelled by action parameters. The variables of a process are private and therefore cannot be accessed by other processes. Like a process, the system is modelled by a state transition system as well. The vector of the process variables forms the system states. The system transitions are described by system actions. In a system action a subset of the processes execute simultaneously a joint action while the other processes perform a stuttering step.

The process *XTPService* in Fig. 2 is a typical example of a cTLA system composed from processes. The processes combined to the system are

```
PROCESS XTPService (XTPCap : Nat) ! XTPCap : capacity of the service
  PROCESSES
    ...;
    C   : Corruptions (Byte,{ (k,k) | k ∈ Byte })
                                ! No Corruptions of data are allowed
    Cap : Capacity (XTPCap) ! Buffersize in number of SDUs
    Id  : SDUId
           ! Assignment of unambiguous sequence numbers
    G   : Gaps (0);          ! No Gaps in transfered data stream
    LIn : LiveIn (...); ! Data units are delivered lively
    ...;
  ACTIONS
    Rq (krq : key; d : Byte) ≜
    ! Transmission of user data d with seq. no. krq
      Id.Rq (krq) ∧ C.Rq (krq, d) ∧ Cap.Rq (krq) ∧
      G.stutter ∧ LIn.stutter ∧ ...;
    fIn (krq : key; d : Byte) ≜ ...;
    nIn (krq : key; d : Byte) ≜ ...;
END
```

Figure 2: Service specification *XTPService*

[1]The data type `key`, which specifies the set of available sequence numbers, is declared in process *Symbols*.

declared in the section `PROCESSES`. For instance, the process C (cf. Fig. 1) is an instance of the process type *Corruptions* with the parameter setting `(Byte,{ (k,k) | k ∈ Byte })`. In the section `ACTIONS` of the system description the system actions are declared by conjunctions of process actions and stuttering steps. In the example *XTPService*, the local process actions `Rq` of the processes *Id*, *C*, and *Cap* are coupled to the system action `Rq`, while the processes R, G, and *LIn* participate in `Rq` by stuttering steps[2].

cTLA supports the superposition (cf. [4, 30]) which is a special kind of compositionality. The superposition guarantees that a property fulfilled by a process or a subsystem is also a property of each system which contains this process or subsystem. It is essential for the concept of the transfer protocol framework, and, in particular, for the reduction of the protocol verification into subsystem implications. In cTLA, the superposition is guaranteed since the composition of processes to systems corresponds to the consistent logical conjunction of processes. With respect to safety properties, the processes are not contradictory due to the private process variables. With regard to liveness properties, composed systems are also not contradictory, since in cTLA only so-called conditional fairness assumptions are used. In contrast to [1], the WF and SF constructs require that a process action has to be performed only, if it is enabled locally as well as it is tolerated by the joint actions of the process environment. Conditional fairness statements are sufficient for the expression of absolute liveness properties if they are joined with the assumption that fair actions are not too often blocked by the environment of a process. This assumption is called an environment condition. Its proof is an integral part of the application of liveness theorems.

3 Transfer Protocol Framework

The transfer protocol framework consists of specification modules and of theorems. The specification modules are modelled by cTLA process type definitions. They describe service constraints, protocol mechanisms, and constraints of a basic transfer medium which is used by the protocol mechanisms. The specification modules are structured into three layers: the Service Contraints (SCs) model single properties of a communication service. As pointed out in the upper part of Fig. 3, one can develop service specifications by composing SC instances. Protocol specifications are described

[2]Stuttering steps are described by the pseudo action name `stutter`.

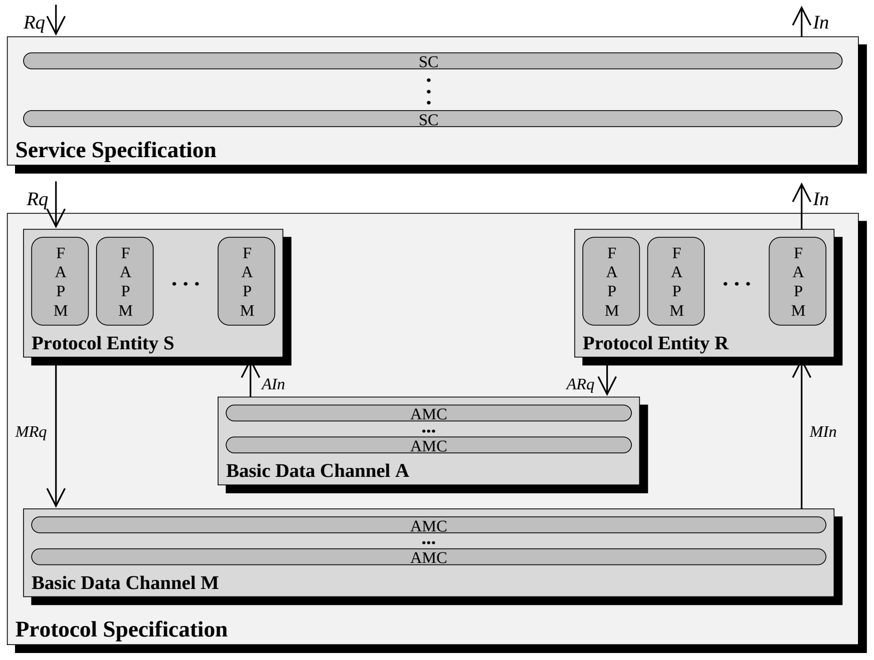

Figure 3: Structures of a service and a protocol specification

according to the well known scenario shown in the lower part of Fig. 3. Like SCs, the Abstract Medium Constraints (AMCs) model properties of the basic transfer medium. The protocol instances are composed from protocol mechanisms. To simplify the protocol verification, the framework provides two different groups of specification modules to describe protocol mechanisms. The Abstract Protocol Mechanisms (APMs) specify abstractions of real protocol mechanisms. They model only the essential functions of the protocol mechanisms, but do not attach importance to details of an efficient implementation. In contrast, the Finite Abstract Protocol Mechanisms (FAPMs) model real protocol mechanisms in a quite direct manner. A real data transfer protocol is specified by composing appropriate FAPMs and AMCs. By the composition of APMs and AMCs, one creates a more abstract protocol which is used as an intermediate model in order to decompose the verification into two phases.

The framework theorems correspond to logical implications between cTLA systems. Due to the structuring of the specification modules into the three layers service, abstract protocol, and protocol, the theorems are divided into two groups. The SC theorems contribute to proofs that an abstract protocol fulfills a service. They guarantee that abstract protocol subsystems, modelled by APMs and AMCs, implement single SCs. The APM theorems are used to prove that a protocol fulfills an abstract protocol. They state implications between a protocol subsystem combined from FAPMs and AMCs and an APM.

Fig. 4 refers to an SC theorem stating that an abstract protocol subsystem *Sys* implies the SC *LiveIn*. Among other APMs and AMCs, *Sys* contains the APMs *SLiveMRq* and *RLiveMRq*. This implication is only valid if the parameter condition *Pars* is true. It ensures that the actual parameters of the process instances of *Sys* and of *LiveIn* are consistent with each other. Furthermore, the whole abstract protocol has to fulfill the invariant condition *EnvCond*, which states that the fair actions in *Sys* are not blocked too often by processes of the abstract protocol specification. Thus, it is guaranteed that the conditional fairness assumptions of the actions in *Sys* fulfill the liveness property to be modelled by *LiveIn*. *EnvCond* has to be checked only if the SC to be fulfilled models a liveness property. To enable a mechanized check of *EnvCond* by COAST, all processes of the framework, which do not violate the environment condition of a theorem, are listed in the section `CORRESPONDS WITH`.

Actually, the framework consists of 133 specification modules (28 SCs, 44 APMs, 14 AMCs, and 47 FAPMs) and 165 theorems (31 SC theorems and 134 APM theorems).

```
THEOREM LiveIn

 LET Pars ≜ mla = { (p,q) | skey[spci[p] ] = skey[spci[q] ] ∧
                            stack[spci[p] ] = stack[spci[q] ] ∧
                            stcre[spci[p] ] = stcre[spci[q] ] ∧
                            p ∈ encpdu ∧ q ∈ encpdu } ∧ ...;
     Sys ≜ SLiveMRq (pdu, pci, usd, encpdu, spci, skey, sack,
                     snak, scre, stack, stcre, skk, skn, skm,
                     usdsize, usdsplit, mcr, rcc) ∧
           RLiveARq (pdu, pci, usd, encpdu, spci, skey, sack,
                     snak, scre, stack, stcre, skk, skn, skm,
                     usdsize, rcu, rcc, ma, mr, mo) ∧
           ... ∧ CC_LiveIn;
     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 ∧ ...;
 IN Pars ∧ Sys ∧ □ EnvCond ⇒ LiveIn (usd, 0, tg, c, {});

 CORRESPONDS WITH
  Process ≜ SBufferKey (pdu, pci, usd, ...);
  Process ≜ SBufferUsd (pdu, pci, usd, skk, ...);
  ...;

END
```

Figure 4: SC theorem to prove the SC *LiveIn*

4 COAST

The selection and arrangement of suitable framework theorems to perform protocol verifications is supported by the tool COAST (Consistency of a specification in cTLA$^+$) [8]. The input files of COAST are specifications of a more detailed system (e.g., a protocol specification) and of a more abstract system (e.g., a service specification) which both are modelled by cTLA process compositions (cf. Fig. 2). Furthermore, COAST has access to a database containing theorems in the syntax described in Fig. 4. By selecting theorems from the database and checking that the specifications are composed and parametrized according to the conditions of the theorem, COAST verifies that the detailed specification implies the abstract specification. This is equivalent to proving that the detailed system correctly implements the abstract system in the sense, that the detailed system contains all of the mandatory properties of the abstract system.

Due to the subdivision of the framework theorems into three layers the tool has to be applied twice to perform a complete protocol proof. In one step one checks that an abstract protocol fulfills a service. The protocol developer provides the abstract protocol specification as the detailed system, the service specification as the abstract system, and the database of all SC theorems to COAST. In the other step COAST proves that a protocol fulfills the abstract protocol. In this case the protocol forms the detailed system and the abstract protocol acts as the abstract system. COAST uses the database of the APM theorems.

COAST performs a protocol proof sequentially in four steps (cf. Fig. 5).

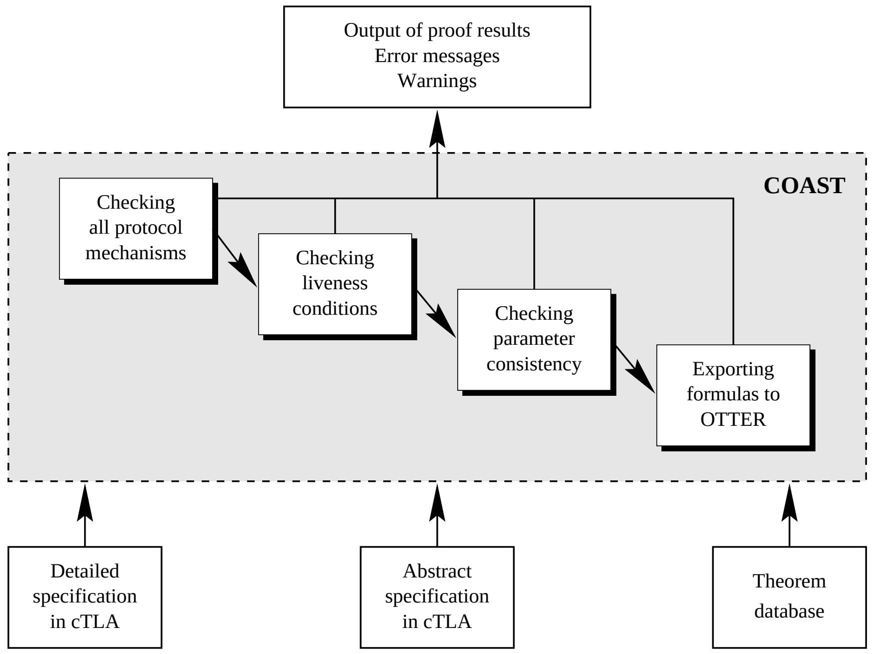
Output of proof results
Error messages
Warnings
COAST
Checking all protocol mechanisms
Checking liveness conditions
Checking parameter consistency
Exporting formulas to OTTER
Detailed specification in cTLA
Abstract specification in cTLA
Theorem database

Figure 5: Elements of the tool COAST

If a check cannot be completed successfully, the tool terminates and reports an error message.

In the first check suitable theorems are selected from the database. As explained in Sec. 3, a theorem guarantees that the subsystem *Sys* of the detailed system implies a process of the abstract system. COAST identifies a theorem for each process of the abstract system. The theorem has to contain the process on the right side of the implication. Thereafter, the tool checks if the processes of the subsystem *Sys* in the theorem are contained in the detailed system. If COAST cannot find a suitable theorem for each process of the coarse-grained specification, the proof will be aborted.

If the process on the right side of a theorem models a liveness property, COAST checks in a second step whether the detailed system contains processes which might spoil the liveness of the subsystem *Sys* and therefore violate the invariant condition *EnvCond* of the theorem. This task is simple since all processes of the framework, which are compatible to *Sys*, are listed in the section `CORRESPONDS WITH` of the theorem (cf. Fig. 4). Thus, COAST checks if the detailed system contains processes which are not contained in this list. If not all processes are compatible, the theorem is rejected and COAST jumps back to the first step selecting a new theorem from the database.

The last two steps deal with the consistency of the actual parameters of the processes. The framework assumes that actual parameters of different process instances are represented by syntactically equal terms, which substitute equally named formal process type parameters. The tool checks this condition in the third step. If it detects syntactically different parameter settings, it adds corresponding proof obligations to the fourth step.

Finally, the fourth step is devoted to the proof of the condition *Pars* of the theorem. *Pars* ensures the logical consistency of the parameter settings and is represented by a first-order logic formula. COAST translates *Pars* (and possibly the additional obligations of the third step) into the input syntax of a frontend tool [13] for the automated theorem prover OTTER [35]. If OTTER verifies *Pars* and the additional obligations, the protocol verification is completed successfully. If OTTER fails to perform a proof, the user of COAST has to decide if the formula is false or if the proof was too difficult or too extensive to be proved without interactive support. Then one can structure the proof and supply additional lemmas to OTTER. Since, however, *Pars* is very simple in the most theorems, OTTER can usually prove these formulas without any further support.

5 Example

For clarification we will outline the proof of the Xpress Transfer Protocol XTP [37, 38]. To support the data transfer requirements of different distributed applications, XTP consists of a broad spectrum of protocol functions (cf. [7, 31]) which are mostly asynchronous to each other. Thus, various combinations of protocol functions are possible. The Transfer Protocol Framework supports this modularity directly. In [23] we introduced the formal specification and verification of XTP. There, the proofs were carried out manually by application of suitable framework theorems. Altogether we needed only three weeks for the specification and verification. However, by application of COAST we can reduce the verification time further.

As outlined in [23], we design a service specification, an abstract protocol specification, and a protocol specification by parametrizing and composing framework processes, first. The service specification *XTPService* is listed

```
PROCESS XTPProtocolAbs
  PROCESSES
! APMs : Protocol mechanisms with infinite variables
    SBK : SBufferKey ! Protocol mechanism modelling the handling of
                     ! sequence numbers in the transmitter entity
              (XTPpdu, ! format of the XTP pdu (modelled by a
                       ! cTLA record)
               XTPpci, ! format of the XTP protocol control
                       ! information
               Byte,   ! XTP provides bytewise data transfer
               ...);
    SBU : SBufferUsd ! Protocol mechanism modelling the store of
                     ! data in the transmitter entity
              (XTPpdu, ! format of the XTP pdu (modelled by a
                       ! cTLA record)
               Byte,   ! XTP provides bytewise data transfer
               ...);
    ...;
  ACTIONS
    Rq (krq : key; d : usd) ≜
    ! Transmission of user data d with sequence number krq to the
    ! service provider
      SBK.Rq (krq,d)  ∧  RBK.stutter  ∧  SBU.Rq (krq,d) ∧
      RBU.stutter ∧ RG.stutter  ∧  RR.stutter ∧
      RD.stutter  ∧  RP.stutter  ∧  ...;
    ...;
END
```

Figure 6: Parametrized Abstract Protocol Specification *XTPProtocolAbs*

in Fig. 2. SCs of the framework (eg., *SDUId*, *Corruptions*, *Gaps*, *LiveIn*) are instantiated and composed according to the desired properties of the service. Since, for example, the service does not tolerate gaps in the stream of delivered data, the specification contains the process instance G of the SC *Gaps*. The process parameter `tg` of *Gaps*, which describes the maximum number of gaps in the stream of delivered data, is set to 0. Thus, gaps are not tolerated at all. The protocol specification *XTPProtocol* is composed from FAPMs, which model the protocol mechanisms of XTP, and AMCs describing the constraints of the basic service. In Fig. 6 the abstract protocol specification *XTPProtocolAbs* is sketched. It models the protocol mechanisms of XTP in a more abstract way than *XTPProtocol* and is composed from APMs (eg., *SLiveMRq*) and AMCs.

Below, we sketch the proof that *XTPProtocolAbs* fulfills *XTPService*. COAST is provided with these specifications and the database of the SC theorems. For instance the theorems listed in Figs. 4 and 7 are contained in this database.

COAST performs the four proof steps outlined in Sec. 4. First, it selects the first process of the service specification. In the example, this is the SC *Corruptions* guaranteeing that corrupted data will not be delivered to the service user. In order to prove this SC, COAST selects the theorem listed in Fig. 7 from the database. In the first step COAST checks that all processes of the subsystem *Sys* are also contained in the specification *XTPProtocolAbs*. Since that is true, the tool completes this step successfully (output

```
THEOREM Corruptions

  LET Pars ≜ mtc ⊆
             {(p,q) | q ∉ encpdu  ∨
                      ( skey[spci[p] ] = skey[spci[q] ] ∧
                       ∀ n ∈ skey[spci[p] ] :
                         (susd[p,n],susd[q,n]) ∈ tc) ) } ∧
            ∀ l,m ∈ usd ∀ n ∈ {0, ..., usdsize[l] - 1}:
              (usdsplit[l,n],usdsplit[m,n]) ∈ tc ⇒ (l,m) ∈ tc;
      Sys  ≜ SBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn,
                         skm, usdsize, mb) ∧
             SBufferUsd (pdu, usd, susd, skk, skn, usdsize,
                         usdsplit) ∧
             RBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn,
                         skm, usdsize, rcu, rcc) ∧
             RBufferUsd (pdu, usd, susd, skk, skn, usdsize,
                         usdsplit) ∧
             MSDUId ∧
             MCorruptions (pdu, mtc) ∧
             MPhantoms (pdu,encpdu) ∧
             CC_Corruptions;
  IN Pars  ∧  Sys  ⇒  Corruptions(usd, tc)

  CORRESPONDS WITH
   ...;

END
```

Figure 7: SC theorem to prove the SC Corruptions

```
* START THEOREM-CHECK !!!! *
Checking Service_Element: Corruptions
- Trying the 1. Theorem for: Corruptions
  TESTING Mechanisms:
  → OK
  TESTING Correspondings:
  → NOT NECESSARY
  TESTING Parameters:
  → OK
→ One or more theorems has been tested correctly for this
   Service_Element !!!!!
...
* END of CHECK !!!! *
```

Figure 8: Output message of COAST

`TESTING Mechanisms` in Fig. 8).

Since Corruptions does not describe a liveness property, the liveness of *Sys* cannot be spoiled by its environment. Thus, COAST omits the second proof step.

In the third step COAST checks, that formal parameters in the theorem, which contain the same name, are replaced by identical defined variables or identical values. For example, the parameters `pdu` of the processes *SBufferKey* and *SBufferUsd* in *Sys* are both replaced by the value `XTPpdu` (cf. 6). COAST finished this check successfully, too (output `TESTING Parameters` in Fig. 8).

In the forth step the condition *Pars* of the theorem *Corruptions* is checked. First, the formal parameters of *Pars* are replaced according to the instantiations of the processes in the specifications *XTPService* and *XTPProcotolAbs*. Thereafter COAST translates the formula into the syntax of the OTTER frontend. Fig. 9 depicts the translation of the first conjunct of *Pars* in the theorem. In the part `FORMULAS` the name *form_Corruptions2_1* is assigned to the first conjunct of *Pars*. The proof script guiding OTTER is defined in the part `THEOREM`. OTTER shall prove the formula *form_Corruptions2_1* by contradiction assuming the already proven formulas *mtc_pred*, etc.

Likewise, COAST performs the checks for each process of *XTPService* and creates OTTER proof scripts. Finally, OTTER verifies the formulas *Pars* of all selected theorems by application of the proof scripts. Since for each process of *XTPService* at least one theorem was identified which passes the checks by COAST and since the OTTER proofs succeeded, the

```
MODULE Corruptions2_equal
FORMULAS
     form_Corruptions2_1 ≜ mtc ⊆ {(p,q) | q ∉ encpdu ∨
                              ( skey[spci[p] ] = skey[spci[q] ] :
                                (susd[p,n],susd[q,n]) ∈ tc) };

THEOREM Test_form_Corruptions2_1
<1>{1}ASSUME mtc_pred, q_pred, p_pred, encpdu_pred, skey_pred,
               spci_pred, susd_pred, n_pred, tc_pred
PROVE form_Corruptions2_1
QED BY CONTRADICTION;

END Corruptions2_equal
```

Figure 9: Part of the OTTER formula for the SC-theorem *Corruptions*

verification that *XTPProtocolAbs* fulfills *XTPService* is successful.

COAST selected 33 theorems of the database of SC theorems which passed the first proof step. Since the abstract protocol specification of XTP is compatible to these theorems with respect to liveness properties, they passed the second step as well. The theorems passed also the third step, since the actual parameters were always replaced by syntactically equal terms. OTTER could prove the *Pars* condition of 25 theorems directly. Eight theorems had to be enhanced with additional data-definitions in order to be proven by OTTER. By this support, OTTER could prove another four theorems. The remaining four OTTER theorem proofs had to be supplied by some simple intermediate lemmata. These lemmata, however, could be designed easily.

For the proof that the protocol system fulfills the abstract protocol system, COAST selected 52 theorems of the 134 theorems of the database of APM theorems. Due to the similarity of the protocol specification *XTPProtocol* and the abstract protocol specification *XTPProtocolAbs*, COAST and OTTER performed these proofs without further support by the user. Altogether, the complete proof that the protocol specification *XTPProtocol* fulfills *XTPService* could be performed within three hours.

6 Conclusion

With the help of the XTP example we outlined the concept of the transfer protocol framework and, in particular, the verification tool COAST. Besides XTP and some simpler sliding window protocols, we applied the framework to specify and prove the complex high-speed protocol MSP [32], too, which could be examined also with a remarkable few expense of work (cf. [24]). The framework can be accessed via WWW (http://ls4-www.informatik.uni-dortmund.de/RVS/P-TPM).

Currently, we expand the specification technique cTLA in order to rise further its acceptance in industrial protocol development projects. On the one hand, cTLA was extended to specify realtime properties and continuous behaviour [22]. This facilitates the formal specification and verification of distributed realtime systems. In particular, it is adapted to control systems for hybrid chemical systems [18]. This cTLA expansion can be utilized, however, for the development of communication protocols (fi. modelling the transmission of multimedia data) as well.

On the other hand we develop a method to translate cTLA system specifications into the hardware description language VHDL [34]. By design compilers (fi. Synopsis) VHDL hardware module descriptions can be transformed automatically into hardware circuits. In this approach we are using cTLA as a system-level description language with the ability of proving the specification against several constraints (the service specification) [9]. The goal of this extension is the transformation of already proven descriptions into hardware circuits. Thus, expensive functional testing of circuits and simulations in order to check the correctness of the specified systems can be saved. This work is funded by the postgraduate research programme CINEMA of the German research foundation DFG.

References

[1] B. Alpern and F. B. Schneider, “Defining liveness”, Information Processing Letters, vol. 21, 181–185 (1985).

[2] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber, “The design of distributed systems — an introduction to FOCUS”, Technical Report TUM-I9202, Technische Universität München, Institut für Informatik, Postfach 202420, D-8600 München (1992).

[3] S. Budkowski, “Estelle Development Toolset”, Computer Networks and ISDN Systems Journal, Special Issue on FDT Concepts and Tools, vol. 25, no. 1, 63–82 (1992).

[4] K. M. Chandy and J. Misra, “Parallel Program Design — A Foundation”, Addison Wesley (1988).

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications”, ACM Transactions of Programming Languages and Systems, vol. 8, no. 2, 244–263 (1986).

[6] E. M. Clarke, O. Grumberg, and D. E. Long, “Model checking and abstraction”, ACM Transactions of Programming Languages and Systems, vol. 16, no. 5, 1512–1542 (1994).

[7] W. A. Doeringer, D. Dykeman, M. Kaiserswerth, W. Meister, H. Rudin, and R. Williamson, “A Survey of Light-Weight Transport Protocols for High-Speed Networks”, IEEE Transactions on Communications, vol. 38, no. 11, 2025–2039, (1990).

[8] O. Drögehorn, “Ein Werkzeug zum formal basierten Entwurf von Hochleistungstransportprotokollen” (in German), Diploma Thesis, Universität Dortmund, Informatik IV, D-44221 Dortmund (1996).

[9] O. Drögehorn, O. Terhorst, H.-D. Hümmer, and W. Geisselhardt, “Formal specification and verification of transfer-protocols for system-design in VHDL”, in: Proceedings of the First International Forum on design languages (FDL), Lausanne (1998).

[10] U. Engberg, P. Grønning, and L. Lamport, “Mechanical verification of concurrent systems with TLA”, in: Proceedings of 4th International Workshop on Computer-Aided Verification, CAV ’92, eds. G. v. Bochmann and D. K. Probst, Lecture Notes in Computer Science 663, Springer-Verlag, 44–55 (1992).

[11] J.-C. Fernandez and L. Mounier, “ ‘On the fly’ verification of behavioural equivalences and preorders”, in: Proceedings of 3rd International Workshop on Computer-Aided Verification, CAV ’91, Lecture Notes in Computer Science 575, Springer-Verlag, 181–191 (1991).

[12] S. J. Garland, J. V. Guttag, and J. J. Horning, “Debugging Larch shared language specifications”, IEEE Transactions of Software Engineering, vol. 16, no. 9, 1044–1057 (1990).

[13] A. Geist, “Eine Theorembeweiser-gestützte Entwicklungsumgebung für die halbautomatische Verifikation verteilter Systeme” (in German), Diploma Thesis, Universität Dortmund, Informatik IV, D-44221 Dortmund (1994).

[14] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, “Simple On-the-fly Automatic Verification of Linear Temporal Logic”, in: Protocol Specification, Testing, and Verification XV, Warsaw, eds. P. Dembiński and M. Średniawa, Chapman & Hall, 3–18 (1995).

[15] W. W. Gibbs, “Software’s Chronic Crisis”, Scientific American, vol. 271, no. 3, 72–81 (1994).

[16] Z. Har’ El and R. P. Kurshan, “Software for analytical development of communications protocols”, AT&T Technical Journal, vol. 69, no. 1, 45–59 (1990).

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

[18] P. Herrmann, G. Graw, and H. Krumm, “Compositional Specification and Structured Verification of Hybrid Systems in cTLA”, in: Proceedings of the 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC98), Kyoto, IEEE Computer Society Press, 335–340 (1998).

[19] P. Herrmann, T. Kraatz, H. Krumm, and M. Stange, “Automated Verification of Refinements of Concurrent and Distributed Systems”, Research report 541, Universität Dortmund, Informatik IV, D-44221 Dortmund (1994).

[20] P. Herrmann and H. Krumm, “Compositional Specification and Verification of High-Speed Transfer Protocols”, in: Protocol Specification, Testing, and Verification XIV, Vancouver, eds. S. T. Vuong and S. T. Chanson, Chapman & Hall, 339–346 (1994).

[21] P. Herrmann and H. Krumm, “Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations”, in: Protocol Specification, Testing, and Verification XV, Warsaw, eds. P. Dembiński and M. Średniawa, Chapman & Hall, 171–186 (1995).

[22] P. Herrmann and H. Krumm, “Specification of Hybrid Systems in cTLA+”, in: Proceedings of the 5th International Workshop on Parallel & Distributed Real-Time Systems (WPDRTS’97), Geneva, IEEE Computer Society Press, 212–216 (1997).

[23] P. Herrmann and H. Krumm, “Modular Specification and Verification of XTP”, Telecommunication Systems, vol. 9, no. 2, 207–221 (1998).

[24] V. Hinz, “Formale Spezifikation und Verifikation eines Hochleistungstransferprotokolls mit dem Transferprotokollframework” (in German), Diploma Thesis, Universität Dortmund, Informatik IV, D-44221 Dortmund (1998).

[25] G. J. Holzmann, “Algorithms for Automated Protocol Verification”, AT&T Technical Journal, 32–44 (1990).

[26] G. J. Holzmann, “Design and validation of protocols: a tutorial”, Computer Networks and ISDN Systems, vol. 25, 981–1017 (1993).

[27] ISO, “ESTELLE: A formal description technique based on an extended state transition model”, International Standard ISO/IS 9074 edition (1989).

[28] ISO, “LOTOS: Language for the temporal ordering specification of observational behaviour”, International Standard ISO/IS 8807 edition (1989).

[29] ITU, “SG X: Recommendation Z.100: CCITT Specification and Description Language SDL” (1993).

[30] Reino Kurki-Suonio, “Hybrid Models with Fairness and Distributed Clocks”, in: Lecture Notes in Computer Science 736, Springer-Verlag, 103–120 (1993).

[31] T. F. La Porta and M. Schwartz, “Architectures, features, and implementation of high-speed transport protocols”, IEEE Network Magazine, 14–22 (1991).

[32] T. F. La Porta and M. Schwartz, “The MultiStream Protocol: A highly flexible high-speed transport protocol”, IEEE Journal on Selected Areas in Communications, vol. 11, no. 4 (1993).

[33] L. Lamport, "The Temporal Logic of Actions", ACM Transactions on Programming Languages and Systems, vol. 16, no. 3, 872–923 (1994).

[34] R. Lipsett, E. Marschner, and M. Shahdad, "VHDL — the language", IEEE Design & Test, 28–41 (1986).

[35] W. W. McCune, "OTTER 3.0 Reference Manual and Guide", Technical Report ANL-94/6, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439-4801, USA (1994).

[36] A. Mester and H. Krumm, "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, Aarhus, BRICS (1995).

[37] Protocol Engines, Incorporated, "XTP Protocol Definition Revision 3.4" (1989).

[38] XTP Forum, 1394 Greenworth Place, Santa Barbara, CA 93105, USA, "XTP Transport Protocol Specification, Revision 4.0", available via FTP: dancer.ca.sandia.gov in directory pub/xtp4.0 (1995).